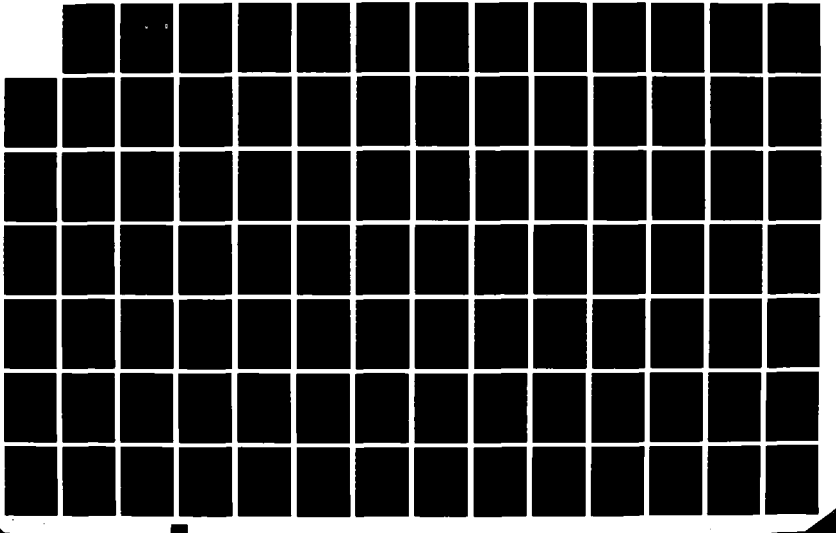
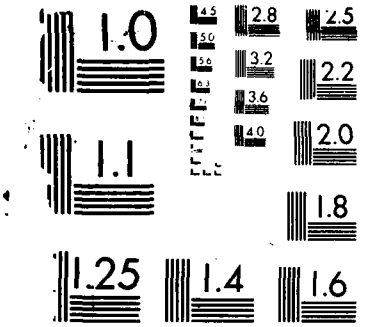
AD-A186 593 DESIGN IMPLEMENTATION AND EVALUATION OF AN OPERATING SYSTEM FOR A NETWORK OF TRANSPUTERS(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA M D CORDEIRO 1/2

UNCLASSIFIED F/G 25/2 NL

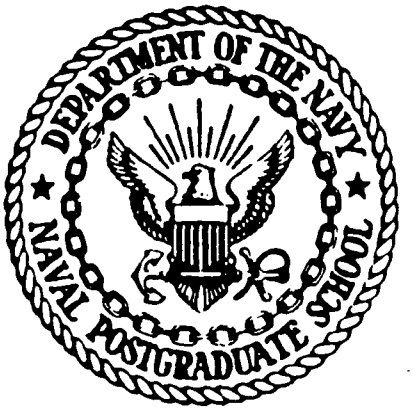




MICROCOPY RESOLUTION TEST CHART

(2)

NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

DESIGN, IMPLEMENTATION AND EVALUATION
OF AN OPERATING SYSTEM
FOR A NETWORK OF TRANSPUTERS

by

Mauricio de Menezes Cordeiro

June 1987

Thesis Advisor Uno R. Kodres

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

A156 593

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT: Approved for public release; Distribution is unlimited.
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION: Naval Postgraduate School; 6b OFFICE SYMBOL (If applicable): 52	7a NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code): Monterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code): Monterey, California 93943-5000
8a NAME OF FUNDING / SPONSORING ORGANIZATION; 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO; PROJECT NO; TASK NO; WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification): DESIGN, IMPLEMENTATION AND EVALUATION OF AN OPERATING SYSTEM FOR A NETWORK OF TRANSPUTERS

12 PERSONAL AUTHOR(S): CORDEIRO, MAURICIO DE MENEZES

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year Month Day)	15 PAGE COUNT
Master's Thesis	FROM ____ TO ____	June 1987	163

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Transputer, Operating System, OCCAM, TDS, Network, Concurrent Processing, Distributed System, Routing, Message Communications

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis presents the **Design, Implementation and Evaluation of an Operating System for a Network of Transputers,** with main focus on the Communication Subsystem. It also introduces the novice to the Transputer Development System (TDS), and suggests a sequence for developing applications.

All the programs and examples presented in this thesis were implemented in the OCCAM1 Programming Language, and using the Transputer Development System (TDS-D600), running under the VAX/VMS Operating System at the Naval Postgraduate School (NPS).

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL: Prof. KODRES, UNO R.	22b TELEPHONE (Include Area Code): (408) 646-2197 — 22c OFFICE SYMBOL: Code 52Kr

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE

Approved for public release; distribution is unlimited.

Design, Implementation and Evaluation of an Operating System for a Network of Transputers

by

Mauricio de Menezes Cordeiro
Lieutenant, Brazilian Navy
B.S., Brazilian Naval Academy, 1976

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1987

Author: Mauricio de Menezes Cordeiro

Approved by: Uno R. Kodres, Thesis Advisor

Daniel L. Davis, Second Reader

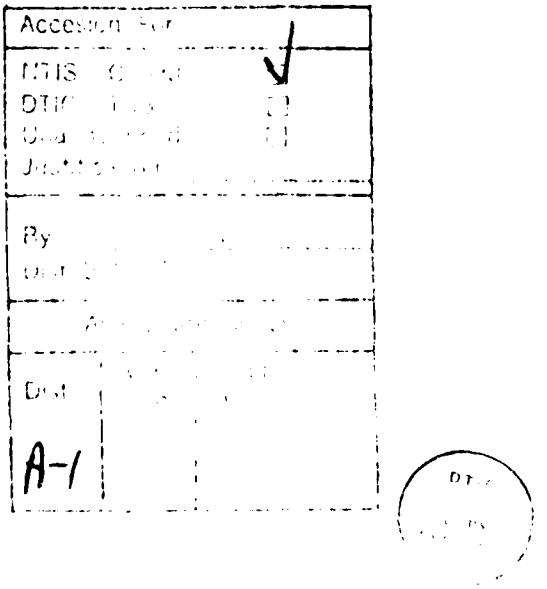
Vincent Y. Lum, Chairman,
Department of Computer Science

Kneale Thomas Marshall,
Dean of Information and Policy Science

ABSTRACT

This thesis presents the **Design, Implementation and Evaluation of an Operating System for a Network of Transputers,** with main focus on the Communication Subsystem. It also introduces the novice to the Transputer Development System (TDS), and suggests a sequence for developing applications.

All the programs and examples presented in this thesis were implemented in the OCCAM1 Programming Language, and using the Transputer Development System (TDS-D600), running under the VAX/VMS Operating System at the Naval Postgraduate School (NPS).



THESIS DISCLAIMER

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

Many terms used in this thesis are registered trademarks of commercial products. Rather than attempting to cite each individual occurrence of a trademark, all registered trademarks appearing in this thesis are listed below the firm holding the trademark:

Digital Equipment Corporation, Maynard, Massachusetts
- VAX 11/780 Minicomputer
- VMS Operating System
- VT-220 Terminal
- VT-100 Terminal

Digital Research, Pacific Grove, California
- CP/M 86 Operating System

INMOS Group of Companies, Bristol, UK
- Transputer
- Occam
- INMOS
- IMS T414
- IMS T800
- TDS
- OPS

Intel Corporation, Santa Clara, California
- iSBC 86/12A Single Board Computer
- Multibus
- 8086 Microprocessor

Microsoft Corporation, Bellevue, Washington
- DOS Operating System

Xerox Corporation, Stanford, Connecticut
- Ethernet

Zenith Data Systems Corporation, St. Joseph, Michigan

Z-248 Microcomputer

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

Dedico esta tese à minha esposa Cristina e aos meus filhos Igor e Lucas, pelo amor, compreensão e carinho dispensados durante toda esta dura jornada.

Aproveito a oportunidade, para expressar todo o meu amor e reconhecimento a meus pais Franklin e Helena, sem os quais esta tese e eu proprio não existiriamos.

À minha sogra Maria, o meu especial e sincero muito obrigado, por todo o apoio e carinho dispensados à minha familia, por ocasião do nascimento de meus dois filhos.

To my thesis advisor, Professor Uno Kodres, I would like to thanks for all the confidence and support, which has never stopped, even when he was passing through some health adversities.

Finally, I would like to send a special thanks to the Technical Staff in the Computer Science Department, especially to Michael Williams, Walter Landaker, Russell Wallen and Rosalie Johnson, for all the support we were given throughout this thesis.

I. INTRODUCTION

A. BACKGROUND

1. The AEGIS Project

The research interests of the NPS AEGIS project embraces a broad spectrum of topical areas within the Computer Science Department. Initially found in the late 1970's it had the primary mission of investigating alternative architectures for the AEGIS Combat System, which are being deployed on board of the U.S. Ticonderoga class (CG-47), whose central unit is the 3D Phased Array Radar AN/SPY-1A.

The basic thrust of this research is the belief that the same software system running under the old and expensives AN UYK-7 computers could run equally well, if not more efficiently, in the commercially available VLSI microprocessors.

A sequence of projects have culminated in the successful Real Time Cluster Star Architecture (RTC*).

The RTC* is a multiple microprocessor system with a hierarchical bus structure resembling the Carnegie Mellon Cm* architecture. RTC* is specifically suited for the development and implementation of real time, concurrent sensor data gathering, display and control systems, which are some of the typical applications in a Weapons System [Ref 1].

Presently, the RTC* is composed of two clusters, each containing four INTEL Single Board Computers based on the 8086 microprocessor. These single boards have from 64K up to 128Kbytes of dual port dynamic RAM being shared among each cluster, with part of this memory space being virtually shared between clusters. All the boards are connected to the INTEL Multibus through an interface control logic unit and the communication between clusters is done via an ETHERNET link.

The software system to support the RTC* was done in parallel with the hardware design and after six years of iterative engineering, refinement and extensions, it evolved to the E-MCORTEX operating system, which was integrated in 1984 as a system software layer over the multiuser CP/M 86 operating system [Ref. 2: p. 10].

As time progresses, the old AN/UYK-7's in the AEGIS system are being replaced by the new AN/UYK-43's, and as expected, in probably less than one decade they will not be capable of handling the increasing demand for some more complex software systems.

That is why the NPS AEGIS Modeling Project, trying to keep up with all the upcoming new technologies, has added to its Laboratory a network of eighteen transputers, which can be very easily connected in various configurations, to allow the user to evaluate and compare them, in a performance basis with the RTC* architecture.

2. Transputer Review

The term transputer is an acronym for "transistor computer" where it reflects the ability of this device to be used as system's building block, much like the transistor was in the past. The nice feature of the transputer is that it adds a new level of abstraction, which provides a very simple way to design concurrent systems.

As a formal definition we could state that a transputer is a single chip microcomputer with its local memory and with four independent links for connecting one transputer to another. The links may be thought of as small special purpose processors which steal no cycles from the main cpu, in such a way that we could have all four links and the cpu working at the same time, without degrading the performance of the program's execution [Ref. 3].

The interprocess communications are done through channels, using a strictly message passage schema where shared memory is not allowed. Each link provides two channels, one in each direction. A message is transmitted as a sequence of bytes and the way the transputers know when the other transputer is ready to receive a message is as follows: the first transputer to become ready transmits the first byte of the message and once it arrives in the other end, it is stored in the buffer of that link, and just when that link is ready to receive the next byte an acknowledge signal is sent back. Each of the links must maintain a buffer of one byte long for this purpose.

The communications between links is bytewise asynchronous and not phase sensitive, but it is, obviously, bitwise synchronous, otherwise we could not sample the bits correctly.

a. The processsor and its scheduler

The transputer, IMS T414, is a general purpose 32 bit microprocessor with a maximum throughput of 10 MIPS.[1] It is highly optimized to implement the OCCAM Programming Language and it has a reduced instruction set, where many of the instructions are one byte long.

---

[1]It depends on the type of the transputer, more specifically on the internal clock under which it is running. The following values apply: T414-12 (6 MIPS), T414-15 (7.5 MIPS) and T414-20 (10 MIPS).

The processor supports two priority levels, high and low, and for each of them it keeps a queue of ready processes. The low priority processes will run only when there are no high priority processes in the queue.

The OCCAM parallel construct is implemented on a single transputer, by timeslicing the processes which are ready at any instant in time. A process is descheduled if it has to wait for communications, timer input or if it completes processing. Another possibility for descheduling, valid only for low priority processes is when its timeslice is finished, so that the next in the queue will be activated. Each timeslice period lasts for approximately 800 microseconds.

b. The T414 Timer

The resolution of the timer depends on which board we are talking about. On the B001 the timer has a resolution of 1.6 microseconds per tick, while in the B003 we have 1 microsecond for the high priority processes and 64 microseconds for the low priority ones. If working with the VAX-VMS the timer ticks every 100 nanoseconds, but it is updated just every 10 milliseconds.

The value obtained from the timer is a signed integer which wraps around at MAXINT ($2^{31} - 1 = 2147483647$) and MININT ($-2^{31} = -2147483648$), so that attention is needed when trying to subtract times.[2] See Figure 1.1 for a summary.

	Resolution	*Half-Cycle*
B001	1.6 usec/tick	57.3 min
B003 (High)	1.0 usec/tick	35.8 min
B003 (Low)	64.0 usec/tick	38.2 hrs
OPS (VAX-VMS)	100.0 nsec/tick	3.6 min

Figure 1.1 T414 and OPS Timers.

c. Memory

The T414 can directly access a linear address space of up to 4 Gbytes. The 32 bit wide memory interface uses multiplexed data and address lines and provides a data rate of up to 25 MBytes/sec.

[2]A routine called tick.to.time will be provided in the O.S. Library Routines, such that all the cases will be handled properly.

There is 2Kbytes of on chip memory which provides a maximum data rate of 80 Mbytes/sec and can be shared among different users through the internal system bus. The latter value is obtained when using a memory with access time of 50 nanoseconds, but it also varies from transputer to transputer.

The address space of the T414 is signed and byte addressed. It ranges from #80000000 which is equivalent to MININT, up to #7FFFFFFF which is MAXINT. The first 2K of memory, in other words, from #80000000 up to #80000800 reference on chip memory, where the first 72 bytes are reserved for system purposes. See Figure 1.2.

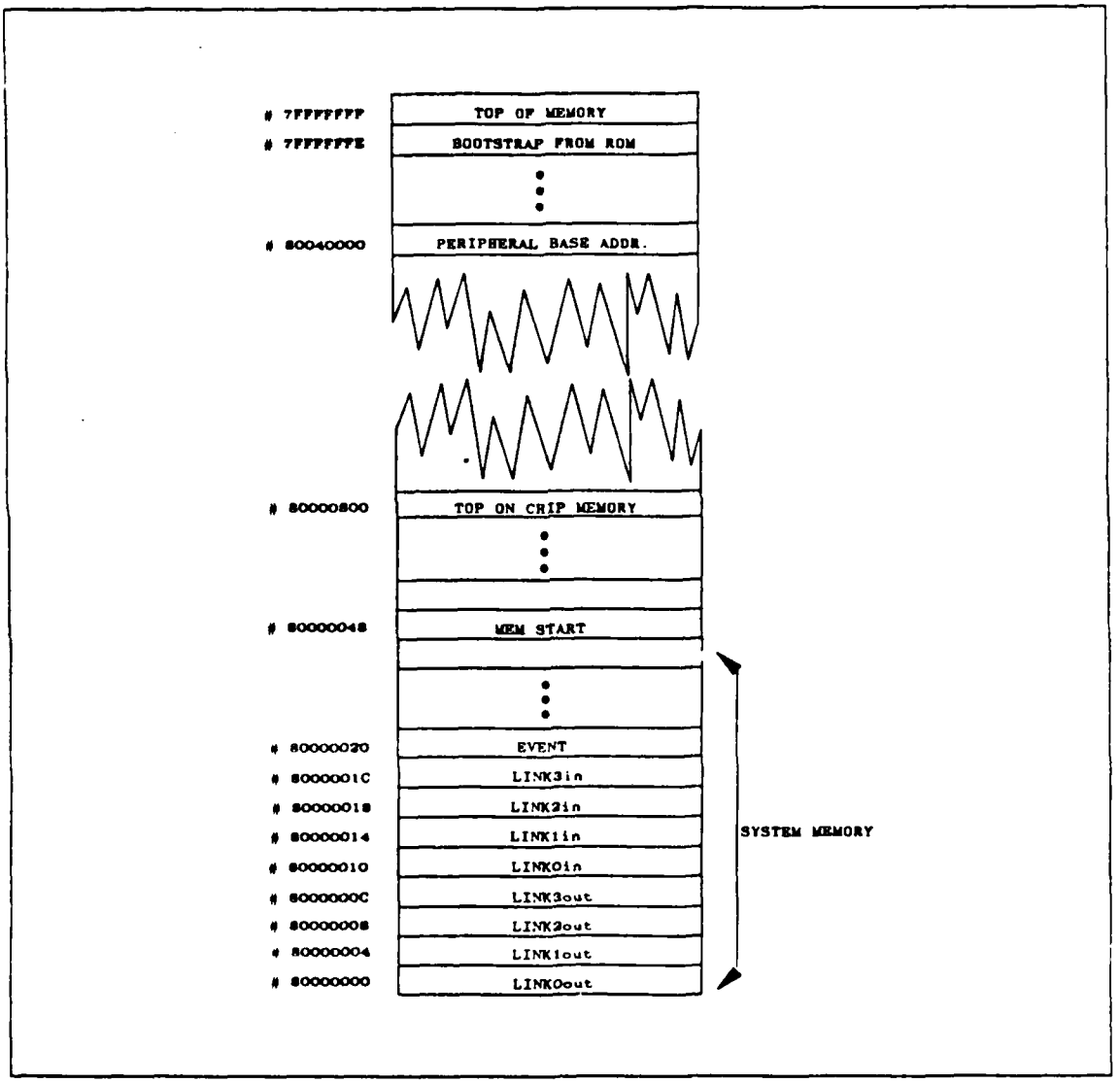


Figure 1.2 T414 Memory Space.

d. Links

The T414 has four full duplex standard links, each providing two unidirectional channels. The links can be thought of, as described earlier, as a special purpose processor which has some DMA block transfer capabilities.

The speeds of the links may be selectable from 10 Mbits/sec or 20 Mbits/sec on the B003 boards, with no choice other than the standard 10 Mbits/sec on the B001 board. The B003 board has the additional capability of maintaining link 0 at 10 Mbits/sec while the remaining links 1, 2 and 3 are at 20 Mbits. Therefore, care must be taken to enforce that both links connecting the B001 and the B003 board are working at the same speed, 10 Mbits/sec.

3. **The Transputers at NPS**

As far as hardware goes, we have in our Lab a Transputer Evaluation Module with four boards B003's, each containing four 32 bit transputers T414-15 (15 MHz) plus 256Kbytes of dynamic RAM per transputer. The fifth board we have is the B001 with a 32 bit transputer T414-12 (12.5 MHz), 64K of dynamic RAM and 128Kbytes of EPROM containing the bootstrap loader, the memory test and the transparent mode software. This board is directly connected to the host computer (VAX/VMS in our case) through a RS-232 serial port and it also provides an additional port for attaching one monitor.

We also have another board which is the B004, which is placed in one of the slots of a personal computer Zenith 248. This B004 board contains a 32 bit transputer T414-15 (15 MHz) and comes with 2Mbytes of dynamic RAM on board. Its basic function is to provide an interface between the PC and the network of transputers, but it also allows us to run programs in its transputer, much likely the B001. For additional information about all the above mentioned boards, please refer to their respective user's manual [Refs. 4,5,6].

It is important to notice that the B003 board does not allow one to have access to the links 2 and 3 of any of its transputers. They come in a fixed configuration (see Figure 1.3), where the only links the user can connect however he desires are the links 0 and 1.

At present we have three software packages on which we can either simulate or actually generate code for the transputer. They are:

- OCCAM Progamming System (OPS) which runs under the VAX/VMS Operating System and allows one to simulate the transputer environment, using the OCCAM1 as the primary language. The code generated by the OPS

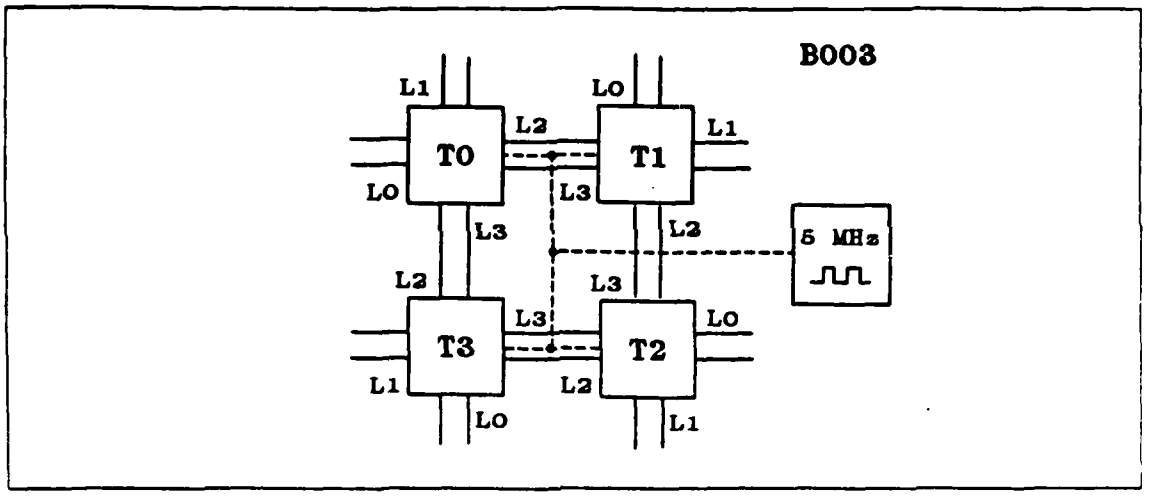


Figure 1.3 B003 board and its fixed connectivity.

compiler is for the VAX/VMS, so that no valid time measurements can be made, nor can we run truly multiprocessor programs. As it stands right now it is just a very good tool for teaching purposes, since it allows many users to run and test their programs, concurrently. Another use of the OPS would be in the early stages of the design, for checking the correctness of some modules, before running them on the transputer itself.

- Transputer Development System (TDS-D600) which also runs under the VAX/VMS Operating System, and whose compiler generates transputer code which can be later on extracted and downloaded into a transputer network. One of its differences from the OPS is the configuration part, where a program can be configured to run in various processors, which are connected in some specified way. The primary language is still OCCAM1.
- Transputer Development System (TDS-D701) which is very similar to the D600, although more powerful, and it runs on an IMS B004 board in collaboration with a small program running under the DOS Operating System in a personal computer, which provides access to the PC's resources. Its primary language is OCCAM2 which has data types, floating point arithmetic, among many other things that are not provided in OCCAM1.

B. PURPOSE OF THIS THESIS

Since this is one of the first thesis to make use of the transputer hardware,[3] our mission was to create a user friendly environment, with all the software necessary for future users to develop their application programs.

---

[3]We had two previous thesis on transputers, but they were actually designed to run under the OPS in the VAX, since we had no transputers at the time they were written [Refs. 7,8].

The tools we are about to describe embraces a library with all the basic I/O routines, such as output to the screen, input from the keyboard, capability of formatting the screen and to write and read from VMS files among others. Also we have developed some utility routines which will allow anyone to dump parts of memory and to get the real time in a readable format anywhere in the program.

However, the central focus of this thesis is on the design and implementation of a basic Communications Operating System, which would make it easier to program a distributed network of transputers. All the effort was made to carry out this task and after many, many changes, we ended up in a very simple and effective design. We are not claiming that this is the only one or the best way of doing it, but it is our hope that it serves as a firm foundation for future and more enhanced implementations.

We also evaluate what is the overhead imposed in the program's execution time, when running under the Operating System, which constitutes one of the most important concerns when dealing with real time systems.

Unfortunately, when this thesis was started we didn't have the OCCAM2 version available to use as our primary language, which would have made life much easier. As a result we are using PROTO OCCAM or OCCAM1 throughout the entire thesis, which is a very simple but primitive language, with no data types, no channel protocols, no floating point arithmetic, etc....

As an auxiliary learning tool we will provide for the novice user of the Transputer Development System for the VAX/VMS, a quick explanation of all its features, its required program structure, its drawbacks and all the points we found obscure in the manuals, whose knowledge would have saved us a lot of hours of reading.

C. THESIS ORGANIZATION

Chapter II begins with a brief overview of the Transputer Development System, in order to assist the reader in understanding its basic features. Next, we suggest a sequence for developing applications, where we present a very thorough description of all the steps involved. Still in this Chapter, we develop a very simple methodology for configuring a network of transputers. The remainder of Chapter II is devoted to some general suggestions, in order to make the working environment, as friendly as possible.

Chapter III describes all major design decisions we had to make, in order to implement the Operating System. The main purpose in doing that, is to provide the

reader with a precise conceptual understanding of the system, which would enable him to perform some major changes in the system, if it is so needed. It also presents a general block diagram of the Operating System.

Chapter IV describes the implementation of the modules in the Operating System. The Library Routines are also covered, mainly the "send" and "receive" routines. A complete guide explaining how to use the routing table is also addressed.

Chapter V evaluates the performance of a program running under the operating system. All the evaluation is done in a comparison basis with the one made by Vanni J.F. in his thesis [Ref. 9], where the transputer is completely evaluated. In this Chapter, we also perform the evaluation of the operating system, when handling multiple hop communications. At the end of Chapter V, we measure the effect of the header size on the transfer rates.

Chapter VI basically describes how to use the Operating System, under the user's point of view. The required program structure is also presented, as well as some hints in how to program with the operating system.

Chapter VII is the final chapter, which includes the conclusions and some suggestions for follow-on work.

Appendices A and B includes the global definitions to be used in either OPS or TDS.

Appendix C contains the file LIBRARY.TDS, with all the available routines to be used in TDS, without using the operating system.

Appendix D contains the source code for the Operating System in the root transputer, while Appendix E contains the remote version of it, in other words, the one which is to be run in remote transputers.

Appendix F describes the evaluation program used to evaluate the Operating System, and it also serves as a sample example on how to use the operating system.

II. A QUICK TDS TUTORIAL

A. WHAT IS TDS ?

The name TDS stands for "Transputer Development System" and it is basically built around the concept of "folding".

Its fold editor is the principal interface between the system and the host computer. It allows the user to insert, edit and delete Occam source text, and to save this text into a VMS file.

Besides its general and standard editing functions, it also contains a set of ten utilities and three special functions, which perform extended tasks with a TDS program.

We will now cover the basics of its folding system, describing all the available commands. We hope that by now the reader has already been exposed to the editor tutorial, where all the basics about "folds" is covered. It is also important to notice at this point, that this editor uses a very unusual sequence of keystrokes and therefore it is of primary importance to have the correct terminal driver running under it. We will assume hereafter that the system we are using is the TDS for the VAX and that our terminal is the VT-100 or VT-200 (in VT-100 mode), but if that is not the case, please refer to the TDS Installation Manual [Ref. 10: Section 1].

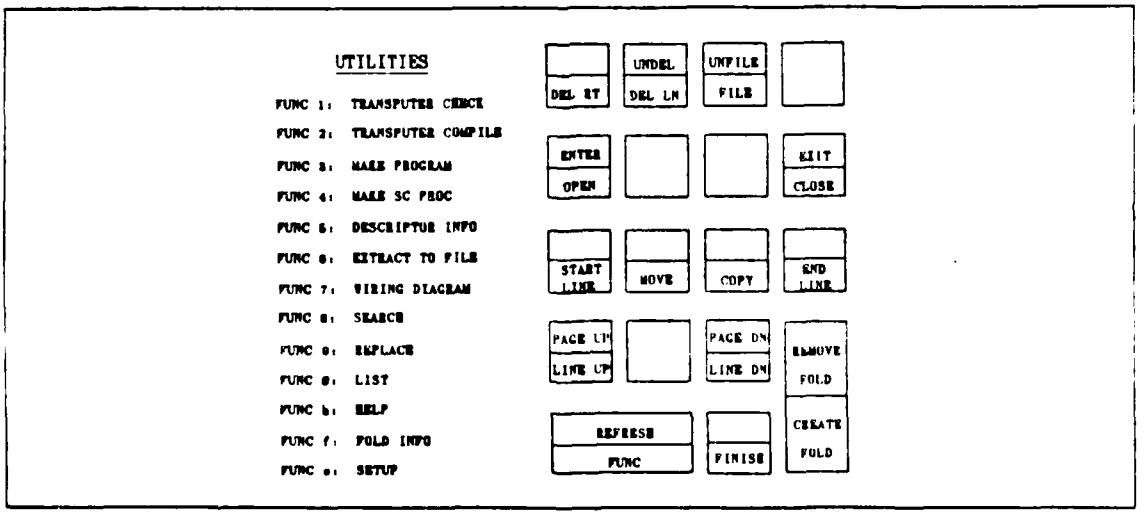


Figure 2.1 DEC VT-100 Keyboard Layout.

Besides all the editing features common to all editors, the TDS has in addition what we call "utilities", which are the following:

- Utility 1 (TRANSPUTER CHECK) - It checks the syntax of occam programs, as well as the consistency of variables and channels used inside PAR constructs. When dealing with more complex structures like for example nested PARs, etc...very often we will have to turn off the "UsageCheck" which is found inside its parameters fold, otherwise it will give us all sorts of error messages.
- Utility 2 (TRANSPUTER COMPILE) - It compiles PROGRAMs and SCs PROCs or it may configure an Occam program to run in a network of transputers. In addition to the same checking performed by Utility 1, it also generates code for the transputer, placing it into a fold. Actually, it generates two folds: the descriptor and the code folds. It shares the same parameters fold with Utility 1.
- Utility 3 (MAKE PROGRAM) - It produces a compilation fold marked as a main program fold. It should be used only in the outer fold to specify the whole program to be downloaded into the network. Typically we will have inside such a fold all the SC folds for each of the transputers being used by that program, plus the configuration fold which carries all the information regarding the connectivity of the network.
- Utility 4 (MAKE SC PROC) - It produces a compilation fold marked for separated compilation. All the processes to be run in a specific transputer must be placed inside a SC, which will be eventually allocated to that transputer in the configuration part.
- Utility 5 (DESCRIPTOR INFO) - Provides information about any SC fold. It uses the descriptor fold to get information such as entrypoint, program size, etc....
- Utility 6 (EXTRACT TO FILE) - It extracts the compiled code that lies inside the "code fold" generated by the compiler and exports it to a VMS file. There is one parameter fold which prompts the user to enter with a filename to which to export that code. The default filename is "ops.tcd".
- Utility 7 (WIRING DIAGRAM) - This utility creates a fold with a textual description of all the link interconnections needed for the configuration specified in that program. This utility is, indeed, very helpful when setting up your link connections.
- Utility 8 (SEARCH) - Searchs for a string from the actual cursor position up to the end of the fold on which it was applied. It doesn't allow the use of any wildcard characters.
- Utility 9 (REPLACE) - Replaces the string we are searching for, by another string. It shares the same parameters fold with the searching utility.
- Utility 0 (LIST) - Produces a printable listing of the contents of a fold and places it into a VMS file. It prompts the user to enter with a filename.

Besides the above utilities we have three more special functions which are:

- Func h (HELP) - Displays a list of all ten utilities provided by the TDS, with a brief description.
- Func f (FOLD INFO) - Displays the type of the fold and its contents.
- Func s (SETUP) - Allows the user to change any of the parameters fold already instantiated with new values.

Once we have gone through this brief description of what TDS is, we should now have the feeling that TDS is very closely related to its fold system. Unlikely other systems where we have a physically separated editor, compiler and linker, in the TDS we have all in one. Also another good point about this approach is that if you get an error while compiling you will be placed right at the error in editing mode, and once ready just call the right utility to compile it again !

The way this editor handles external files is also very unique. What we have to do is just to open a fold, name it with the filename and extension of the file we want to be attached to this fold, press the file key PF3 and that is it. That is how it does the job of linking almost transparent to the user.

Just for the sake of completeness, it is worth mentioning the system files which are used by the TDS:

- TDSVT100.OBJ - Transputer Development System for VT-100 terminals.
- TDSVI920.OBJ - Transputer Development System for the TVI-920 terminal.
- TDSTABLE.OBJ - Transputer Development System with table-driven terminals.
- OPSKRNL.OBJ - TDS Kernel which is identical to the OPS kernel.
- TDSSETUP.COM - It is a VMS command file which sets up the TDS environment. Must be executed in the beginning of every session.

B. STRUCTURE OF A TDS PROGRAM

In this Section we will cover the basic structure of a TDS program when running without the Operating System, which will be covered in later Chapters. Any program intended to run under TDS, in other words, in a transputer network, must have a well defined structure, which doesn't allow much freedom for changes (see Figure 2.2).

The basic idea is that for each different process to be run in a different transputer, we must make it a separately compiled unit. The number of parameters depends on how many hardware links are being used by that process, and also if any constants are coming as parameters from the configuration part. As we already know,

```
PROGRAM progname
  SC transputer.1 (CHAN A,B,C,D,E,F,G,H)
    PROC transputer.1 =
      ... global definitions
      ... library routines
      ... PROC terminal.driver
      ... PROC user.1
      PAR
        terminal.driver
        user.1:

  SC transputer.2 (CHAN A,B,C,D,E,F,G,H)
    PROC transputer.2 =
      ... global definitions
      ... library routines
      ... PROC user.2
      SEQ
        user.2:

          o
          o
          o

  SC transputer.n (CHAN A,B,C,D,E,F,G,H)
    PROC transputer.n =
      ... global definitions
      ... library routines
      ... PROC user.n
      SEQ
        user.n:


  ... configuration declarations

  PROCESSOR 1
    ... channel placements
    transputer.1 (...placed channels...)
  PROCESSOR 2
    ... channel placements
    transputer.2 (...placed channels...)

          o
          o
          o

  PROCESSOR n
    ... channel placements
    transputer.n (...placed channels...)
```

Figure 2.2 Program Structure in TDS.

the B003 board has some links which are hardwired, providing no access to them. These channels need not be placed in the configuration.

Inside each SC we should create a fold with the most used definitions and declarations (see Appendix B). Similarly, the library fold (see Appendix C) should

contain some often needed routines such as I/O routines and other utilities. Our suggestion is that all useful routines should be included in this fold, as they are created. The approach we have taken is to make them filed folds in such a way that whenever you make a new program, all you have to do is create two new folds and attach those files to them.

The sequence of steps to attach these files in our program is the following:

1. Make sure you have these files in your working directory.
2. Open a fold inside the program you are working on.
3. Name this fold with the name of the file you want to attach.
4. File this fold by pressing PF3 on the VT-100 terminal.
5. If you have some limitation in memory or if you are not going to use all the routines and definitions that are in there, you should unfile those folds in order to not interfere with the original contents and proceed with the desired modifications. This step as we can see is an optional step and is just carried out for memory savings and readability purposes.

As depicted in Figure 2.2, the third fold inside the SC PROC is the terminal driver, which is crucial if we are using screen outputs or keyboard inputs. It defines hardware memory locations which represent uart (universal asynchronous receiver-transmitter) registers, such as mode register, status register, command register, etc.... All of these are defined as offsets to the peripheral base address which is #80040000. Its basic functions are to reset the uart which we are going to work with,[4] and to define the baud rate for communications between the processor and the monitor. The first one is accomplished by the procedure reset.uart and since it takes a while for the uart to become ready, a built-in delay is provided inside this procedure.

The terminal driver is always ready either to receive a character typed at the keyboard or send something to the screen. If you check the code it is clear that both tasks are just performed after the uart receives a tx.ready or a rx.ready in the status register. Furthermore, if the uart does not receive either flag within 5.12 seconds, the uart is considered to have failed and the terminal driver is exited without further notice! The reason I am telling you this is because we had some intermittent problems in the very beginning of our research, which were very nasty to isolate, and ended up being a problem in the uart.

---

[4]We have two uarts, the uart A is connected to the terminal and uart B to the host computer.

It is also worth mentioning that unlike OPS, where we must send the "end of buffer" ascii code at the end of the message we are going to output to the screen, in TDS we don't have to.

The terminal driver must be placed in PAR or PRI PAR with the user process in order to work properly. The choice of either one construct is not always clear, and it is intimately related with performance, but the unwary use of it may bring up subtle points when dealing with complex programs with nested PARs and PRI PARs, so that the suggested approach is to make your entire program with no PRI constructs and just after it has been proved correct, you should assign the priorities where needed.

In the PROC so called "user.n", we have a standard structure like any other programming language such as Pascal, PL/I, etc... where we have a declarations part, a bunch of procedures which may be nested at any level and finally the main body of our outer PROC user.n. The only main difference is that we should make the channel placements inside this procedure, attaching the software channels to the hardware links of the particular transputer, to which that process is going to be downloaded. Of course, these placements must be in accordance with the configuration.

As one may notice we have put A, B, C, D, E, F, G and H as channel parameters for the SCs, but rather than calling them generically as we did, we could just as well have put the actual channel's names as parameters. In doing so, we wouldn't have to make their placements inside the PROC user.n, since they were going to be directly related to the order specified in the configuration.

C. RECOMMENDED SEQUENCE WHEN DEVELOPING APPLICATIONS

In this Section we will present a suggested sequence of steps when building applications, which in our understanding provides the best results mainly when dealing with medium to large programs. During this and the next few Sections we will be dealing with the same basic program in order to give you a better global idea of all the steps involved.

For the time being assume that the requirements definition and the functional specification phases are completed and the architectural design is underway with all the modules and interfaces already defined.

At this point since all the main modules with their interfaces are already specified, we can have a good idea of how many processors could we use to map our application, as well as which modules could be placed in different processors.

The experimental network will be as depicted in Figure 2.3 where we have 17 transputers divided into 4 clusters with 4 transputers each, and one root transputer. The main purpose of this program will be to allow the novice OCCAM programmer to understand the structure of a TDS program, as well as how to configure a network of transputers.

In this program the root transputer will be running the so called "hostproc", which basically receives a character typed on the keyboard and broadcasts it to four transputers, one in each cluster. Upon receiving the character, these transputers which will be running the process "route", will route the character to each of the transputers left in that cluster. Finally, all the recipient transputers will echo back the same character to the root transputer, so that at the end of the program we will have 12 characters printed on the screen.

The next phase in the traditional software engineering life cycle is the module design, where all the interfaces between modules should be already defined. The module design is concerned with internal features of the module like algorithms, data structures, etc.... In OCCAM terms, the main goal of the module design should be to implement each module as an SC PROC, where all the communication between modules must be done via channels.

Once we are ready to start developing our modules, we can either use the OPS or the TDS. This choice is not very clear, but seems to us that the OPS provides a nice timesharing environment for the early stages of the design, since we could have many users developing and testing their programs concurrently, under the VAX/VMS operating system.

Once all the module design teams have their programs logically correct and running under OPS, they should be integrated as dictated by the previous architectural design, but still under the OPS, where all the interfaces between modules could be checked and validated against typical inputs. As one can see up to this point, no transputer hardware was necessary, and the reason we are emphasizing this is because if we had chosen the TDS instead, we would certainly have had a bottleneck problem in the usage of the B001 board, since it allows just one user at a time. Another main reason in using OPS lies in the fact that in doing so, we could use the powerful debugging tools running under the VMS operating system.

The next step is a controversial one, where we transform an OPS program into a one-transputer TDS program; it will be entirely covered in the next Section. Although

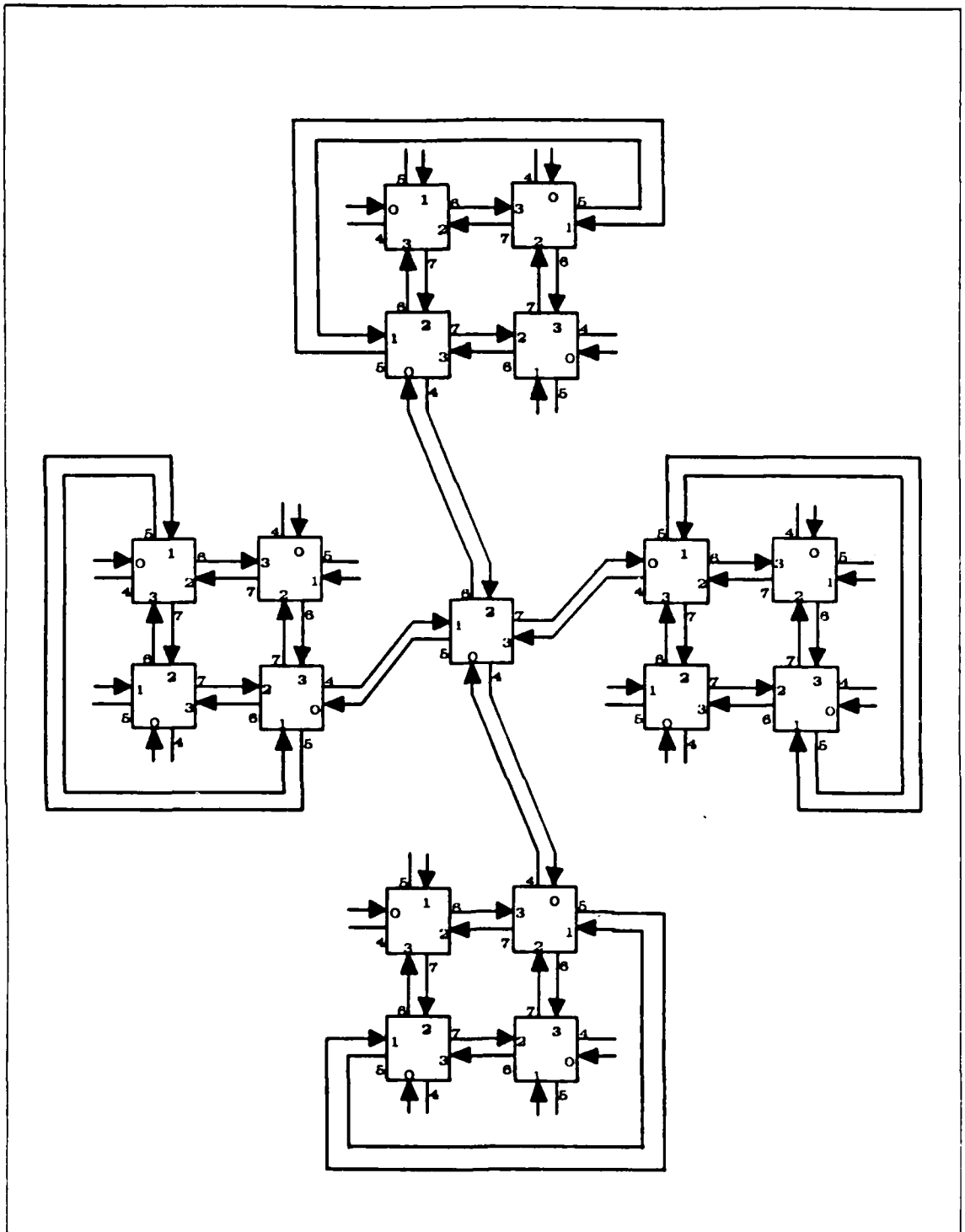


Figure 2.3 A Network with four Clusters.

it looks like a redundant step, I can assure to you that it is not; many bugs can be inserted into the program just by changing global definitions, changing library routines, inserting the now required terminal driver and mainly when trying to use the unique TDS constructs such as BYTE.SLICE.INPUT, WORD.SLICE.OUTPUT, etc... instead of the standard OCCAM channels i/o operations, which are much less efficient (2 to 5 times) than the previous built-in procedures, as fully documented in the Reference 9. Of course this last change need not be done if you don't have any sort of time constraints, otherwise they are crucial, since the differences in time are enormous.

If for anything else, this step should be carried out in OPS, just because we have much better debugging capability than when running in many processors, and keep in mind that any multiprocessor program adds some new potential sources of errors, which are not always easily identified!

Finally we should map this one-transputer program onto a n-transputer program, where this "n" is dictated by the number of modules (SC PROCs) we have, which can be parallelized, and of course by the availability of processors. This conversion process will be described in Section E.

As one can realize, this methodology will not help as far as real time debugging goes, but it will at least provide an effective way to achieve static logical correctness of the program.

D. CONVERTING OPS INTO ONE-TRANSPUTER TDS PROGRAM

According to our recommended sequence for developing applications, there will a point in time when you have developed your program under OPS and want to run it in a single transputer. In these cases you should proceed by checking all the global definitions to see if they are still applicable to a TDS program. for example, the channel Screen in OPS must be placed at "1" and the channel Keyboard at "2", but in TDS this cannot be done, since "1" and "2" will correspond respectively to link1out and link2out addresses. Actually, in TDS the Screen and Keyboard are standard channels which communicate with the terminal driver routine and they don't need to be placed. Those are the basic differences between the global definitions for OPS and for TDS, but for further comparison refer to Appendices A and B, where we present both files. It is important to notice that these global_def.ops, global_def.tds, library.ops and the library.tds files are not required by OCCAM, they constitute just another way of structuring a program, and making it easier to read and maintain.

Now if we look at Figures 2.4 and 2.5 we can see very easily all the steps involved in converting the OPS program.[5] First, as already suggested in the previous paragraph, we should change all the global definitions, as well as the library routines by the TDS equivalents. Second, you should include the terminal driver routine, which is used just in TDS, and place it in parallel with the main user process which was running under OPS. Third, change the PROGRAM fold which is embracing the whole OPS program by an SC fold, otherwise we won't be able to instantiate it in the configuration part.

Finally, you have to do the configuration part, and since we are talking about a program to be run in just one transputer, the configuration becomes extremely simple, where we have only the Processor *number*, followed by the name of the outermost SC PROC, with no channel parameters, since no external communication is going to take place. As you may have noticed, we inserted an additional fold of type PROGRAM embracing the SC and the configuration. This is not necessary, it only allows to compile and configure at the same time, otherwise you will have to apply the "compile utility" in both folds separately.

Once your program is successfully compiled in TDS and it is running properly, you could then try one more refinement step in order to speed up your program, and that is by using the unique TDS constructs like BYTE.SLICE.INPUT, WORD.SLICE.OUTPUT, etc ... instead of the standard OCCAM channels i/o operations like "chan ?" and "chan !". When you are done, compile and run it again.

E. MAPPING FROM ONE TO MANY TRANSPUTERS

Although we recommend to perform the previous step in every program, we understand that the experienced programmer may skip that step for small or even medium programs, but when dealing with more complex programs with intensive communications between processes, it is strongly advised to run it first in one transputer, where you have more debugging capabilities and once it is proven to be logically correct and with no deadlocks, we should map it onto more transputers.

The basic steps to accomplish this mapping are the following:

1. Remove the outermost SC PROC.
2. Find those SC PROCs which have exactly the same code, differing just by the name and merge them into just one SC PROC with a common name.

---

[5]For convenience we have marked with an asterisk all the changed lines in the converted TDS program presented in Figure 2.5.

```
... PROGRAM echo.all
  ... PROC echo.all
    ...F global_def.ops
    ...F library.ops
    ...  SC PROC hostproc (CHAN hostin0,hostin1,hostin2,hostin3,
                           hostout0,hostout1,hostout2,hostout3)
    ...  SC PROC Route00 (CHAN charin,charout,routeto1,routeto2,
                          routeto3,echofrom1,echofrom2,echofrom3)
    ...  SC PROC Route10 (CHAN charin,charout,routeto1,routeto2,
                          routeto3,echofrom1,echofrom2,echofrom3)
    ...  SC PROC Route20 (CHAN charin,charout,routeto1,routeto2,
                          routeto3,echofrom1,echofrom2,echofrom3)
    ...  SC PROC Route30 (CHAN charin,charout,routeto1,routeto2,
                          routeto3,echofrom1,echofrom2,echofrom3)
    ...  SC PROC echochar01 (CHAN charin,charout)
    ...  SC PROC echochar02 (CHAN charin,charout)
    ...  SC PROC echochar03 (CHAN charin,charout)
    ...  SC PROC echochar11 (CHAN charin,charout)
    ...  SC PROC echochar12 (CHAN charin,charout)
    ...  SC PROC echochar13 (CHAN charin,charout)
    ...  SC PROC echochar21 (CHAN charin,charout)
    ...  SC PROC echochar22 (CHAN charin,charout)
    ...  SC PROC echochar23 (CHAN charin,charout)
    ...  SC PROC echochar31 (CHAN charin,charout)
    ...  SC PROC echochar32 (CHAN charin,charout)
    ...  SC PROC echochar33 (CHAN charin,charout)

    ...  main program echoall
         CHAN pipe[32]:

         PAR
           hostproc (pipe[0],pipe[2],pipe[4],pipe[6],
                     pipe[1],pipe[3],pipe[5],pipe[7])
           route00 (pipe[1],pipe[0],pipe[9],pipe[11],
                    pipe[13],pipe[8],pipe[10],pipe[12])
           route10 (pipe[3],pipe[2],pipe[15],pipe[17],
                    pipe[19],pipe[14],pipe[16],pipe[18])
           route20 (pipe[5],pipe[4],pipe[21],pipe[23],
                    pipe[25],pipe[20],pipe[22],pipe[24])
           route30 (pipe[7],pipe[6],pipe[27],pipe[29],
                    pipe[31],pipe[26],pipe[28],pipe[30])
           echochar01 (pipe[9],pipe[8])
           echochar11 (pipe[15],pipe[14])
           echochar21 (pipe[21],pipe[20])
           echochar31 (pipe[27],pipe[26])
           echochar02 (pipe[11],pipe[10])
           echochar12 (pipe[17],pipe[16])
           echochar22 (pipe[23],pipe[22])
           echochar32 (pipe[29],pipe[28])
           echochar03 (pipe[13],pipe[12])
           echochar13 (pipe[19],pipe[18])
           echochar23 (pipe[25],pipe[24])
           echochar33 (pipe[31],pipe[30])
```

Figure 2.4 OPS program.

3. The terminal driver which was in parallel with all the SCs, must now be placed inside the SC PROC that will run in the root transputer.

```
...  PROGRAM echo.all
*  ...  SC PROC echo.all
*    ...F global_def.tds
*    ...F library.tds
     ...  SC PROC hostproc (CHAN hostin0,hostin1,hostin2,hostin3,
                            hostout0,hostout1,hostout2,hostout3)
     ...  SC PROC Route00 (CHAN charin,charout,routeto1,routeto2,
                           routeto3,echofrom1,echofrom2,echofrom3)
     ...  SC PROC Route10 (CHAN charin,charout,routeto1,routeto2,
                           routeto3,echofrom1,echofrom2,echofrom3)
     ...  SC PROC Route20 (CHAN charin,charout,routeto1,routeto2,
                           routeto3,echofrom1,echofrom2,echofrom3)
     ...  SC PROC Route30 (CHAN charin,charout,routeto1,routeto2,
                           routeto3,echofrom1,echofrom2,echofrom3)
     ...  SC PROC echochar01 (CHAN charin,charout)
     ...  SC PROC echochar02 (CHAN charin,charout)
     ...  SC PROC echochar03 (CHAN charin,charout)
     ...  SC PROC echochar11 (CHAN charin,charout)
     ...  SC PROC echochar12 (CHAN charin,charout)
     ...  SC PROC echochar13 (CHAN charin,charout)
     ...  SC PROC echochar21 (CHAN charin,charout)
     ...  SC PROC echochar22 (CHAN charin,charout)
     ...  SC PROC echochar23 (CHAN charin,charout)
     ...  SC PROC echochar31 (CHAN charin,charout)
     ...  SC PROC echochar32 (CHAN charin,charout)
     ...  SC PROC echochar33 (CHAN charin,charout)
     ...  main program echoall
          CHAN pipe[32]:
          PAR
*           terminal.driver (Keyboard,Screen,port,baud)
            hostproc (pipe[0],pipe[2],pipe[4],pipe[6],
                      pipe[1],pipe[3],pipe[5],pipe[7])
            route00 (pipe[1],pipe[0],pipe[9],pipe[11],
                     pipe[13],pipe[8],pipe[10],pipe[12])
            route10 (pipe[3],pipe[2],pipe[15],pipe[17],
                     pipe[19],pipe[14],pipe[16],pipe[18])
            route20 (pipe[5],pipe[4],pipe[21],pipe[23],
                     pipe[25],pipe[20],pipe[22],pipe[24])
            route30 (pipe[7],pipe[6],pipe[27],pipe[29],
                     pipe[31],pipe[26],pipe[28],pipe[30])
            echochar01 (pipe[9],pipe[8])
            echochar11 (pipe[15],pipe[14])
            echochar21 (pipe[21],pipe[20])
            echochar31 (pipe[27],pipe[26])
            echochar02 (pipe[11],pipe[10])
            echochar12 (pipe[17],pipe[16])
            echochar22 (pipe[23],pipe[22])
            echochar32 (pipe[29],pipe[28])
            echochar03 (pipe[13],pipe[12])
            echochar13 (pipe[19],pipe[18])
            echochar23 (pipe[25],pipe[24])
*           echochar33 (pipe[31],pipe[30])    :
*  ... configuration
*    PROCESSOR 0
*      echo.all
```

Figure 2.5 Converting to TDS.

4. The global_def.tds and library.tds files should now be placed inside each of the SC PROCs which are going to be downloaded in different transputers.

5. Change the configuration to run the program in multiple transputers. This step will be covered in full detail in the next Section, so that for the time being we will limit ourselves to write down the header of the fold.

The Step 2 deserves some additional explanation, and that is because when we are trying to map and run a multiprocessor program in just one processor, the only way to simulate very closely the structure of such a program is by making copies of all the procedures that are going to be ultimately downloaded in different processors, name them differently, and finally run them in parallel in the uniprocessor system.

However, when making the final mapping onto more than one transputer, this redundancy is no longer needed, and it should be eliminated, in other words, all SCs containing the same code should be merged into just one, and at loading time the loader will take care of sending one copy for each processor, according to the configuration. Although this last step is not mandatory, we strongly recommend it, because in doing so you will be reducing substantially the code size to be downloaded to the transputer network, increasing the readability of the program as well.

```
... PROGRAM echoall
  ... SC PROC hostproc (CHAN hostin0,hostin1,hostin2,hostin3,
                        hostout0,hostout1,hostout2,hostout3)
  ... SC PROC Route (CHAN charin,charout,routeto1,routeto2,
                     routeto3,echofrom1,echofrom2,echofrom3)
  ... SC PROC echochar (CHAN charin,charout)
  ... configuration
```

Figure 2.6 The Previous Program Mapped onto many Transputers.

F. CONFIGURING A NETWORK OF TRANSPUTERS

Let's start by asking ourselves what is a configuration? Why is it needed ? Well, the configuration is the way we have to specify which process is going to run in which processor and also to map the interprocessor channels onto the hardware processor links. This is accomplished by using some OCCAM1 extensions like PLACED PAR, PROCESSOR *number*, PLACE *channell* AT *address* and CHAN *channel* AT *address*.

The code for any processor must be contained in a single SC PROC and the processor number can be any valid integer, which is just a logical identifier of that processor. However, the first processor to be declared must be always the root transputer, in other words, the processor connected to the host computer, which is the one responsible for bootstrapping and loading the code in the entire network.

Each of the SC PROCs may be instantiated on any number of processors in the network, although it is exported from the host to the root just once. Further copies will be provided and sent by the root transputer to the others in the network.

We have two ways of attaching software channels to hardware links, one is at the program level and uses the CHAN AT statement, and the second is with the PLACE AT statement which is used at the configuration level. The first one is optional, but if we don't use it we must declare the channels explicitly as formal parameters to the SC, and they will be mapped to the actual parameters, at the time that SC is called or instantiated at the configuration level. On the other hand, if we decide to use the CHAN AT statement inside our program, the parameters to the SC PROC can be in any order and can have any name; the only thing that will be checked by the compiler is the match of the number of formal against the number of actual parameters. If you look back in our Figure 2.2 you will notice that we have used channels A, B, C, D, E, F, G and H as formal parameters, what suggest to us that we have to use some CHAN AT statements inside our process "user.n".

If there is a requirement to connect two links from the same processor, a soft channel must be used.

A network configuration can be viewed as a PROGRAM consisting of a collection of SC PROCs which are instantiated from inside some PLACED PAR construct. SCs at this level must have just CHAN or VALUE types as formal parameters.

Let's go now through the configuration of our old program echo.all where all these steps will be made much clearer for you. Usually, after deciding how many parallel processes you are going to have and how many processors you are going to need, the next step is to define how they will be connected in a very broad sense. So, let's suppose we want to run the echo.all program in the network presented in the Figure 2.3.

Once the previous base steps have been accomplished, we suggest the following sequence of steps in order to properly configure a network of transputers:

1. Number all the transputers using a structured ordering schema (see Figure 2.7).

2. Name the channels of the links used[6] to connect the different transputers. We suggest the use of an array of channels because it will allow you to make use of replicators as we will see later (see Figure 2.7).
3. Place the correct process in each of the transputers in the network (see Figure 2.7).
4. Start making the placements for all the transputers in the network, just by reading directly from your sketch (see Figure 2.8).
5. Instantiate the procedures such that the number of actual parameters matches exactly the number of formal parameters in the SC PROC. The order is irrelevant if we are also making the link placements inside the SC PROC.

As demonstrated, the configuration is a very simple matter if we follow the suggested steps, but sometimes when we have more than one processor executing the same process, it is very likely that we will be able to recognize some fixed pattern in their connectivity, which will allow us to simplify the configuration by using some PLACED PAR replicators. That is why in the first and second steps we have suggested to use a structured ordering schema for the transputer number and an array of channels for the channel names. Now it is just a matter of finding a fixed pattern between the channel index, transputer number and its link number. Finally, after some reasoning, we were able to find an equivalent configuration which is showed in Figure 2.9. A further simplification could be to take out the placements of those hardwired links in the B003 board, but this will be left as exercise for the reader.

This extra step is more an adornment than anything else, but it is strongly recommended when dealing with very large networks, because in doing so we will provide a better picture of our entire network. An experienced OCCAM programmer just by looking at the configuration, can have a pretty good idea of the connectivity of the entire network, in other words, if it uses a tree structure, a pipeline, a ring, etc.... This feeling will be almost impossible if the processors are declared one at the time.

Two facts are important in this analysis, the first is to realize that no simplification would be possible if there were no processors running the same process and the second is to understand that this embellishment in the configuration is not mandatory.

One of the points that we have just stated but didn't cover in detail was the division of a program in a parallel number of processes. It is obvious that not every problem can be partitioned into smaller tasks to be carried out by different processors,

---

[6]The channels from the hardwired links in the B003 board, do not need to be placed in the configuration part.

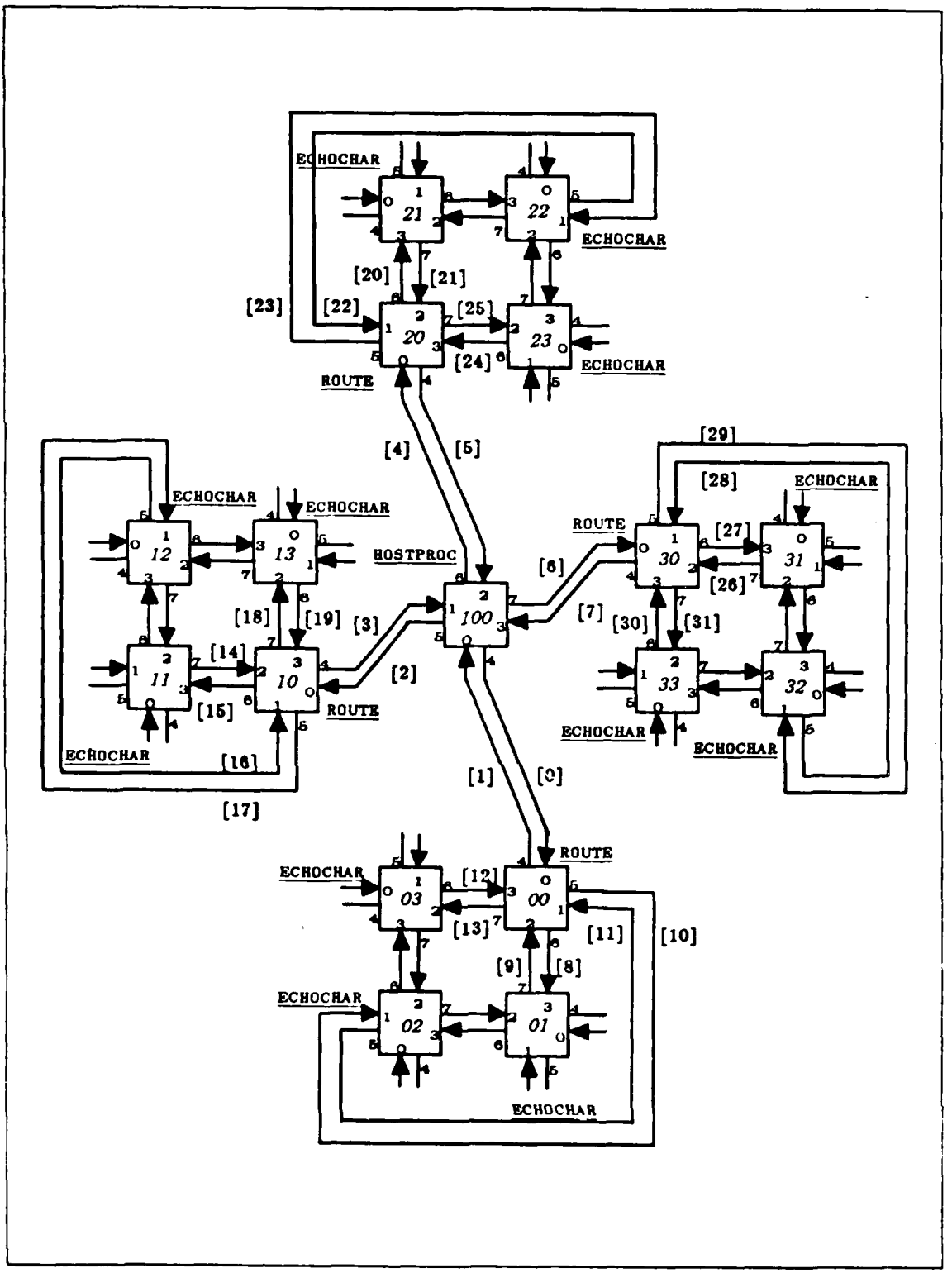


Figure 2.7 Steps 1, 2 and 3 of a Configuration.

```
PLACED PAR
  PROCESSOR 100
    PLACE pipe[0] AT link0in       :
    PLACE pipe[1] AT link0out      :
    PLACE pipe[2] AT link1in       :
    PLACE pipe[3] AT link1out      :
    PLACE pipe[4] AT link2in       :
    PLACE pipe[5] AT link2out      :
    PLACE pipe[6] AT link3in       :
    PLACE pipe[7] AT link3out      :

    hostproc (pipe[0],pipe[2],pipe[4],pipe[6],
              pipe[1],pipe[3],pipe[5],pipe[7])

  PROCESSOR 00
    PLACE pipe[1] AT link0in       :
    PLACE pipe[0] AT link0out      :
    PLACE pipe[10] AT link1in      :
    PLACE pipe[11] AT link1out     :
    PLACE pipe[8] AT link2in       :
    PLACE pipe[9] AT link2out      :
    PLACE pipe[12] AT link3in      :
    PLACE pipe[13] AT link3out     :

    route (pipe[1],pipe[0],pipe[9],pipe[11],
           pipe[13],pipe[8],pipe[10],pipe[12])

  PROCESSOR 10
    PLACE pipe[3] AT link0in       :
    PLACE pipe[2] AT link0out      :
    PLACE pipe[16] AT link1in      :
    PLACE pipe[17] AT link1out     :
    PLACE pipe[14] AT link2in      :
    PLACE pipe[15] AT link2out     :
    PLACE pipe[18] AT link3in      :
    PLACE pipe[19] AT link3out     :

    route (pipe[3],pipe[2],pipe[15],pipe[17],
           pipe[19],pipe[14],pipe[16],pipe[18])


              .
              .
              .


  PROCESSOR 32
    PLACE pipe[29] AT link1in      :
    PLACE pipe[28] AT link1out     :

    echochar (pipe[29],pipe[28])

  PROCESSOR 33
    PLACE pipe[31] AT link2in      :
    PLACE pipe[30] AT link2out     :

    echochar (pipe[31],pipe[30])
```

Figure 2.8 The Complete Configuration.

```
PLACED PAR
  PROCESSOR 100
    PLACE pipe[0] AT link0in       :
    PLACE pipe[1] AT link0out      :
    PLACE pipe[2] AT link1in       :
    PLACE pipe[3] AT link1out      :
    PLACE pipe[4] AT link2in       :
    PLACE pipe[5] AT link2out      :
    PLACE pipe[6] AT link3in       :
    PLACE pipe[7] AT link3out      :

    hostproc (pipe[0],pipe[2],pipe[4],pipe[6],
              pipe[1],pipe[3],pipe[5],pipe[7])

  PLACED PAR j = [0 FOR 4]
    PROCESSOR 10*j
      PLACE pipe[(2*j)+1] AT link0in       :
      PLACE pipe[2*j] AT link0out          :
      PLACE pipe[10+(6*j)] AT link1in      :
      PLACE pipe[11+(6*j)] AT link1out     :
      PLACE pipe[8+(6*j)] AT link2in       :
      PLACE pipe[9+(6*j)] AT link2out      :
      PLACE pipe[12+(6*j)] AT link3in      :
      PLACE pipe[13+(6*j)] AT link3out     :

      route (pipe[(2*j)+1],pipe[2*j],pipe[9+(6*j)],
             pipe[11+(6*j)],pipe[13+(6*j)],pipe[8+(6*j)],
             pipe[10+(6*j)],pipe[12+(6*j)])

  PLACED PAR i = [0 FOR 4]
    PROCESSOR (10*i)+1
    PLACE pipe[9+(6*i)] AT link3in     :
    PLACE pipe[8+(6*i)] AT link3out    :

    echochar (pipe[9+(6*i)],pipe[8+(6*i)])

  PLACED PAR i = [0 FOR 4]
    PROCESSOR (10*i)+2
    PLACE pipe[11+(6*i)] AT link1in    :
    PLACE pipe[10+(6*i)] AT link1out   :

    echochar (pipe[11+(6*i)],pipe[10+(6*i)])

  PLACED PAR i = [0 FOR 4]
    PROCESSOR (10*i)+3
    PLACE pipe[13+(6*i)] AT link2in    :
    PLACE pipe[12+(6*i)] AT link2out   :

    echochar (pipe[13+(6*i)],pipe[12+(6*i)])
```

Figure 2.9 A Simplified Configuration.

but even if they could, at the actual state of the art, there is no automatic machine where we put the entire program as input and the machine would generate an optimal division of processes to be parallelized.

As a conclusion we should mention that this whole configuration procedure is a very simple one, even for very large and complex systems, and furthermore, the program can be developed with little or no thought to such matters and then, the required configuration can be performed after the program logic is proven to be correct. It is in this way that large and complex programs can be written while the actual hardware is still in paper design.

Therefore, the key idea is that configuration does not affect the logical behavior of a program. It only enables a program to be arranged so that its performance requirements are met.

G. CUSTOMIZING YOUR ENVIRONMENT

When dealing with either OPS or TDS, it is extremely important to create a friendly environment to work, otherwise you will spend most of your time performing unnecessary bookkeeping. The main reason for that is because a very big number of files is created for each complete cycle of a program,[7] and since the default filenames for some of the above operations are pretty vague, it is important to define a more strict file naming rule. Some other areas are also affected by the lack of a consistent naming rule, for example, OPS and TDS programs are quite different but both use the OCCAM programming language, so that if we use the traditional file naming rules, we would end up with some name followed by the extension OCC for both programs, which is not recommended by obvious reasons.

For all these reasons we have decided to make up our own file naming rules, which are described in Figure 2.10.

When you apply the utility to get a printout of an OPS or TDS program, all the folds are opened and they come up as "--", which is exactly identical to a comment in OCCAM1, so that to avoid confusion we will always use comments with "---" instead of "--". This way, by looking at the printout, we will be able to very easily differentiate a fold from a comment.

Another decision we had to make was regarding the global definitions and the library routines each program was using. It was really messy to make a program, because we had to pick up routines and definitions from different places, and finally put them inside our program, so that we decided to concentrate all in four files so called global_def.tds, global_def.ops, library.tds and library.ops.

---

[7]For a cycle we mean the phases of editing, compiling, linking, extracting and printing.

```
.TDS →    source code of a TDS program
.OPS →    source code of an OPS program
.OCC →    source code can be either for TDS or OPS
.LST →    printable version of a TDS program
.LIS →    printable version of a OPS program
.TCD →    extracted transputer code (TDS)
.EXE →    executable VAX code (OPS)
.OBJ →    relocatable VAX code (OPS)
.DSC →    descriptor information
.CDE →    non extracted transputer code
```

Figure 2.10 File extensions.

As a final step towards the customization of our environment we have made a login.com file for the VAX/VMS, where most of the commands are PC-like. See Figure 2.11.

```
$ @dra0:[occam]opssetup
$ @dra0:[occam.tdsdir]tdssetup
$ set dir [occam.brasil]/version_limit=20
$ prot :== set protection = (owner:r,group:r,world:r)
$ prot :== set prot = (o:rwed,g:re,w:re)
$ d  :== dir/size=used/width=(filename=28)/columns=2
$ cd :== set default
$ md :== create/dir
$ up :== set default [-]
$ ty :== type/page
```

Figure 2.11 Sample login.com for the VAX/VMS.

III. OPERATING SYSTEM DESIGN

A. WHY AN OPERATING SYSTEM ?

As the program complexity increases and more processors are added to the system, some hardware limitations become more critical and a series of new potential sources of errors are added to the program. In the transputer case for example, the four existent output channels will shortly become a bottleneck due to the increasing demand in communications, forcing the programmer to change the logic of his algorithm to comply with the actual architecture. Another problem that will arise is how to route a message to a non adjacent transputer in the network? How to output to the Screen from a remote transputer ?

As widely known, the main purpose of any operating system is to provide a user with the ability to use the system or a family of systems, without having to know the detailed hardware interconnections scheme for each specific system. In the specific case of the transputer, we have tried to follow this same line of thought, and after some reasoning, we have reached a very simple model for an operating system for a network of transputers. In our model, the user will be able to use simple primitives like "send" and "receive", to perform the necessary intercommunication between processors. The main idea behind this approach is to release the user from the obligation of taking care of the channels placements, and all other implications, which are derived from this latter one. In other words, the user will not need to be concerned in how the message will get there.

Another feature, which was included in our model, is the capability of sending two or more messages in parallel, to the same destination transputer, without having to assign or allocate several hardware links to handle this communication. The final goal is to make this another abstraction to the user, where the operating system would multiplex the different messages through the same hardware link, and this does not imply inefficiency, since the destination transputer would have to handle the messages sequentially anyway, afterall it is still a single processor.

Once we have given sufficient reasons to support our claim, that a sort of basic operating system for a transputer network is vital, let's go into the other Section where we will try to cover all the steps of our design, in a very simple and practical way.

B. THE DESIGN

Because of the fact that transputers have only local memory, a first approach and probably the only one at the current state of the art, was to employ a distributed operating system. An operating system kernel would reside in each node processor to supervise the user processes running on the node and to handle message traffic.

The basic part of our design will be towards building an efficient communications system, but we will also provide some I/O handling, as well as some utilities like getting the real time, dumping memory, etc..., from "any transputer" in the network, which will greatly enhance the overall debugging capability of the network, and it will make it much easier to program.

One of the first design issues to arise was regarding the protocol to be used in our communication subsystem, more specifically, what kind of information should be carried by the message header.

The first needed information, and also the most obvious one, is the transputer id number, which will identify the destination transputer for a message. This number, as we will see later, must be in accordance with the routing table, since it will be used as an index to retrieve information from this table.

The second information to be carried by the header is the message size, since we have decided to support variable length messages. Here we had a trade off between versatility (variable length), and efficiency (fixed length), but in this case we have chosen to go towards the first one.

The third header component is not an obvious one, which is the channel id number. It must be unique[8] in the entire system. This channel id will allow the system to determine within one transputer, which process, and ultimately, which channel is supposed to receive that message.

Therefore, the header which we will be using throughout our system is four bytes long and has the format specified in Figure 3.1.

We could have used an integer value, which is also 4 bytes long, to carry all the header information, but it would take too long to decode it, and besides, the time to output four bytes with the BYTE.SLICE construct is approximately the same as to output an integer [Ref. 9]. The difference in decoding time is because with the byte structured

---

[8]After looking at the implementation, it will become evident that the uniqueness of the channel id is a very important requirement, since otherwise it may lead to dubious results. However, it could be eliminated if we have added another field, the transputer origin, in the header of our protocol.

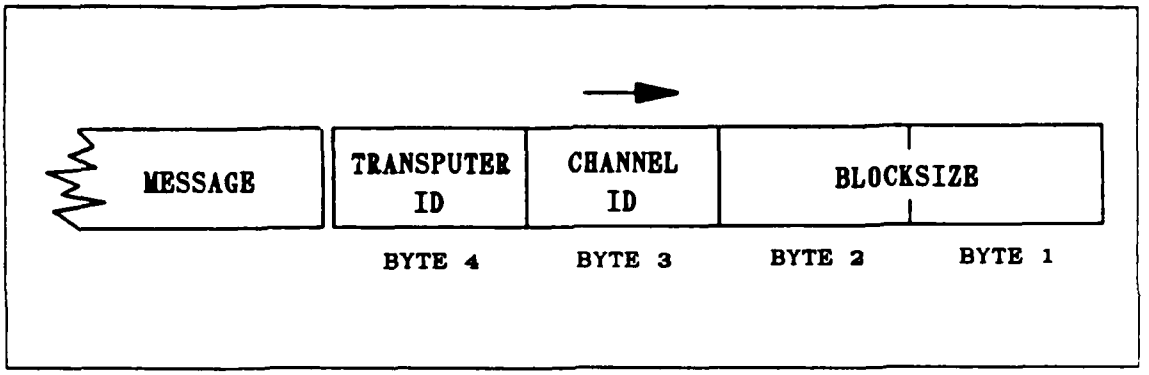


Figure 3.1 The Message Header Format.

header we just have to fetch the proper field and we are done, while with the integer header we will have to perform some additional arithmetic operations like divide, remainder, etc....

In the introductory Section of this Chapter we have discussed some nice features to have in our system, but how could we implement them, keeping the entire process as efficient as we can ? Certainly, the answer at a first glance does not appear to be very simple, since we can have so many different communications paths as depicted in Figure 3.2. For example, one internal process of transputer #X could be trying to communicate with another internal process in the same transputer, or this same process could be willing to talk to a process in transputer #Y, etc... and keep in mind that in the worst case we could have "any number" of internal processes trying to communicate between each other, and "any number" of processes trying to talk to remote transputers through the four output links, and if that is not bad enough, we could have the four input links receiving messages either for some process in this transputer or to be bypassed to some other transputer in the network, and remember that all this could be happening in parallel, except for the internal processes communications which would be done in a timeslice fashion, but they could be still inside a parallel construct.

In the previous paragraph we have said that "any number" of processes could be trying to output through the four output links in parallel, and this statement deserves an additional explanation. In the actual OCCAM implementation this could never be done, because just four software channels could be attached to the four existent output hardware links, and therefore we could have at most four links trying to output at the same time, but since it was a design decision to keep the interface between the user

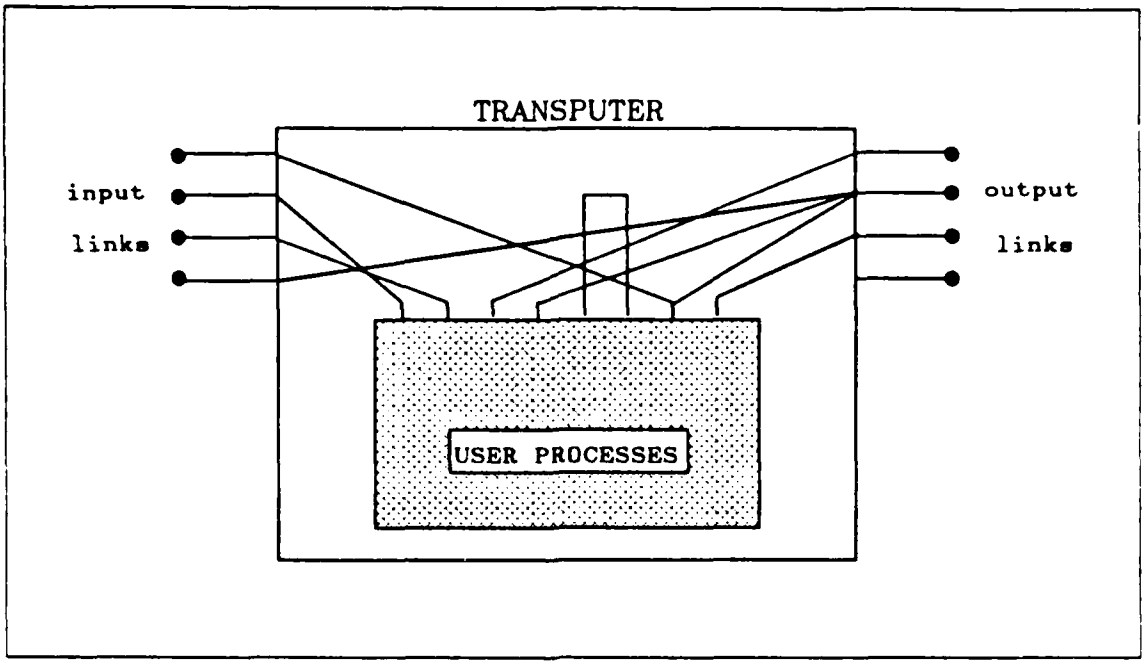


Figure 3.2 The Possible Communications Paths.

processes and the hardware channels as abstract as we could, we are going to implement this extra feature in the Operating System model. For example, suppose we had a case where the algorithm to be implemented had to send two messages in parallel to some transputer X. If we had decided to use straight OCCAM, although "logically" a parallel operation is what we want, in practice the programmer would have to either change the logic of his algorithm because of the above mentioned physical limitations, or he would have to assign a second hardware link to that same transputer X, what is not recommended by obvious reasons. Thus, what we tried to do is to take this preoccupation from the programmer, by building a sublayer of software which would allow any number of output requests to be placed in parallel, even if they have the same transputer as destination.

Let's make up an example, where some transputer is receiving, in parallel, a stream of external values from three distinct transputers. It must calculate their totals and send them to three parallel processes running in another transputer, using the remaining link (see Figure 3.3). The current solution to this problem would be to output the totals in any sequential order with no concern about the order they became ready, in other words, the first total in the sequence could happen to be the longest to calculate and as result we would be blocking the other totals to be sent, delaying the

entire process. With our new approach, the programmer could maintain the algorithm logic by sending them all in parallel, and leave to the operating system the task of multiplexing them through the output link[9] as they became ready. You should argue that we could have used the "ALT" construct and get the same final result, and that is partially true when dealing with non adjacent transputers, but the problem is that it blocks the other processes until that first one is done, therefore, in the extreme case it could even block them for ever. On the other hand, with the "PAR" construct if some process is taking too long, it will timeout and the scheduler will put the next ready process to execute.

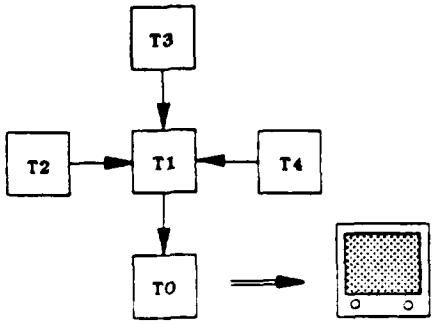
It is also worth pointing out, that when we have many transputers in the network, it becomes much more complex, since we won't know whether or not the final destination is ready to receive that message, so that as a general rule, avoid as much as you can to use time dependent algorithms, because they are very likely to deadlock the system.

Another decision we had to make was regarding the usage of multiple buffers for storing incoming and outgoing messages, which were not ready to be received or delivered. What we were trying to achieve with multiple buffering, was to keep the communication paths free for any messages which might be trying to bypass that transputer, in order to get to its final destination.

However, after some thoughts and after making some rough implementations, we have reached the point where in order to maintain multiple buffers, we would have to lose the parallelism in the input links, because there was no way to get around using the OCCAM1 programming language, and also the overhead imposed to manage the buffers pool seemed to be very large, turning it to be less efficient than with just one buffer. So, in the actual implementation as we will see, any incoming message to a transputer will be stored into the Operating System buffer space, tieing up the channel where it came from, until it is either consumed by some process in that transputer, or bypassed to some other transputer in the network. On the other hand, if it is an outgoing message, it will be kept in the user's memory space of the transmitting transputer, having no effect on the bypassing traffic.

---

[9]Have you noticed in Figure 3.3 that we have used an array of channels in our proposed solution ? The reason for that will become completely clear after reading the implementation Chapter.




```
PAR
  link0in ? stream0              --- from transputer #2
  link1in ? stream1              --- from transputer #3
  link2in ? stream2              --- from transputer #4

... evaluating the 3 totals
```

PROGRAMMER'S INTENTION
(not allowed)

```
PAR
  link3out ! total0              --- to transputer #1
  link3out ! total1              --- to transputer #1
  link3out ! total2              --- to transputer #1
```

CURRENT SOLUTION

```
SEQ
  link3out ! total0              --- to transputer #1
  link3out ! total1              --- to transputer #1
  link3out ! total2              --- to transputer #1
```

OUR PROPOSED SOLUTION

```
SEQ
  chan[50] ! total0              --- to transputer #1
  chan[60] ! total1              --- to transputer #1
  chan[70] ! total2              --- to transputer #1
```

Figure 3.3 An OCCAM Limitation.

As you can see, we will have to deal with all kinds of mutual exclusion problems, since we'll have in most of the cases many more parallel requests than available resources (links), but here is where the transputer architecture, as well as the OCCAM construct "ALTernate", become very handy. The first one by providing a built-in process scheduler with two priority levels, a timer and a memory management unit, and the second by providing a trivial solution to the mutual exclusion problem,[10] as we will see later in the next Chapter.

Now arises another problem, how could we route an incoming message to the correct internal process it is supposed to be routed to ? One solution could be to create in each transputer a channel-id table in memory (see Figure 3.4), where we should have all the channels which communicate with external transputers and one id number associated with each of these channels. Obviously, the id number would have to be carried by each message using that specific channel.

CHANNELS	ID #
out0	10
hostin1	31
from.radar	16
in4	27
⋮	⋮

Figure 3.4 A Sample Channel-id Table.

Another solution that came up after some unsuccessful trials with the channel-id table, and also a much better one in our opinion, is the idea of creating an array of channels, where the id number could be the subscript of the array itself. Simple, is it not ? Also this decision would make our mutual exclusion handler much simpler, as you will see later in the implementation Chapter.

[10] I would like to emphasize that as you may recall, this is one of the most traditional and difficult problems when building operating systems.

As you can see, in our abstraction model we will have our Operating System with the control of all the hardware link communications and with an user interface that will allow the user processes to communicate with the outside world in a very simple way. See Figure 3.5.

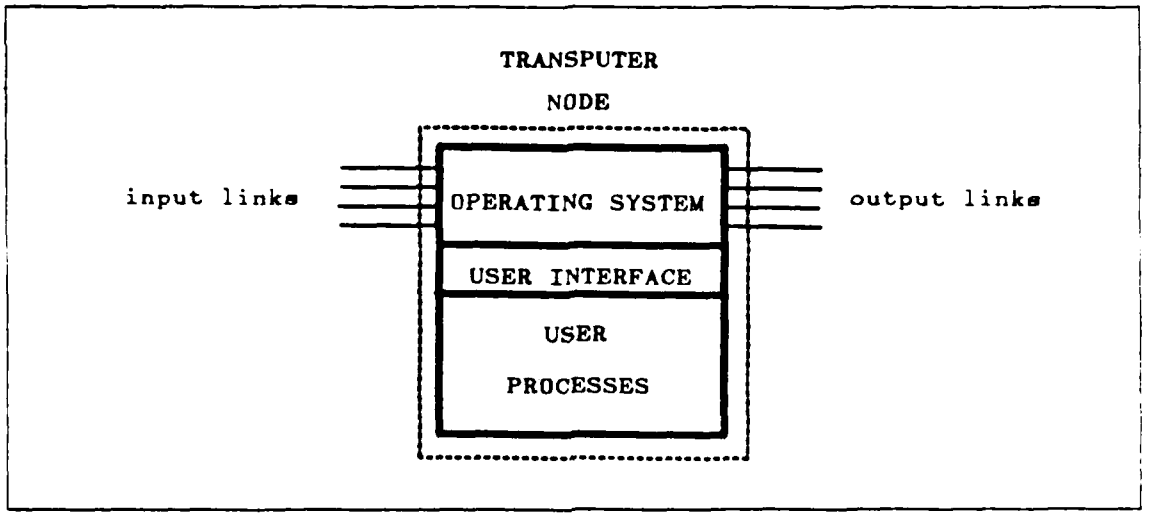


Figure 3.5 User Abstraction.

Therefore, based upon all the previous design decisions, our Operating System will present the following characteristics:

- It will support a maximum of 256 transputers in the network, but since we are using, for convenience, a routing table which supports only 18 entries, it will be limited to 18 transputers. This can be very easily modified, and it will be explained in the Section covering the routing table.
- It will support a maximum of 256 user channels active at one time, for communications with other transputers. Actually this number will drop to 20 channels as we'll see in the implementation Chapter.
- it has no limit to internal soft channels.
- the maximum message length supported is 64 Kbytes.

As we can see, we have much more than we will really need for most typical applications, so that our protocol could be very easily modified and optimized. In Chapter VI we will evaluate two versions of the operating system, one with a 4 byte header, and the other one with 3 bytes, and we will be able to see, very clearly, the effect of the header size in the transfer rate.

Let's now cover more in detail the block components of our Operating System, as well as all the data and control flow that is going on in there. The major blocks which compose our Operating System are the Input Handler, the Screen Handler, the Output Handler and the Operating System Library Routines as depicted in Figure 3.6.

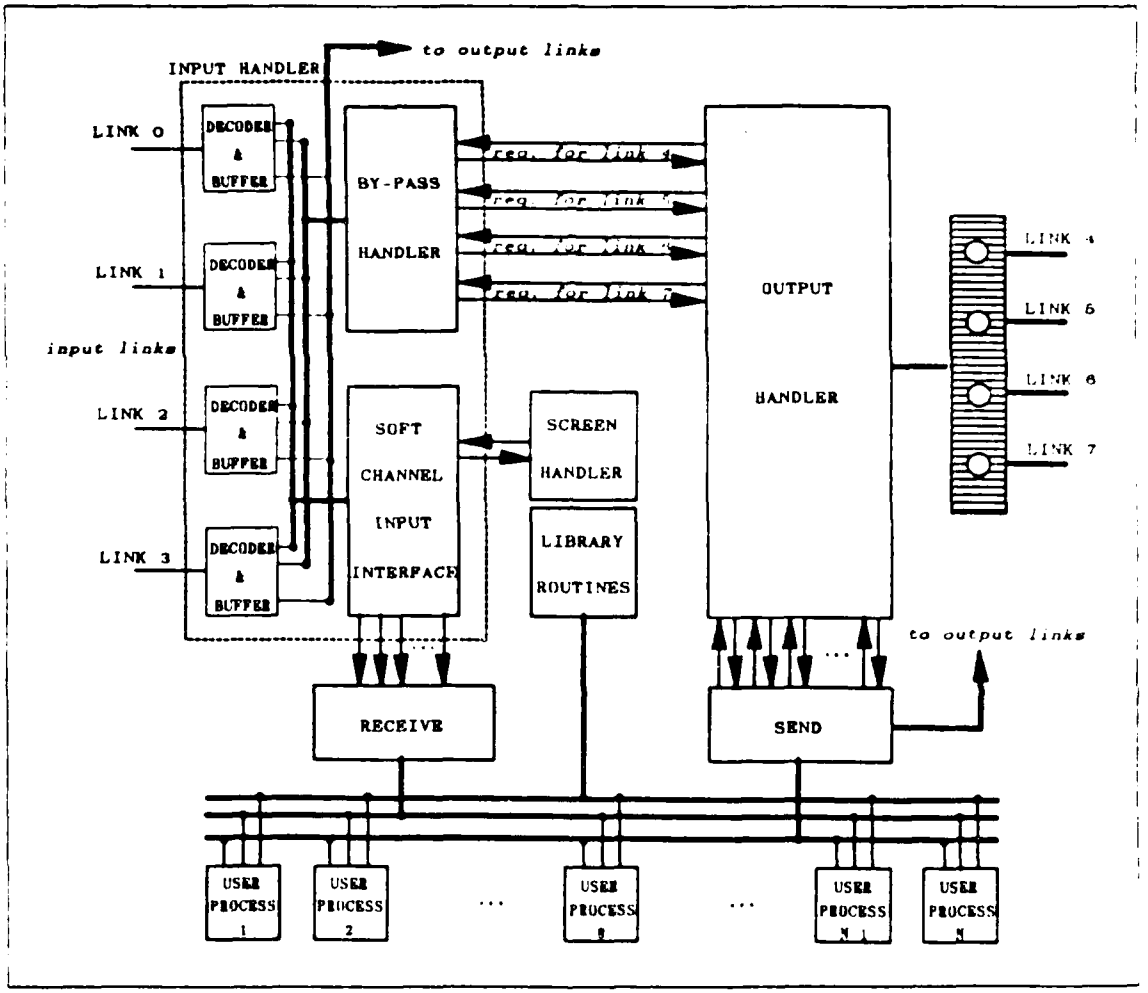


Figure 3.6 Operating System Block Diagram.

1. Input Handler

The Input Handler is composed of three basic blocks, the decoder and buffer, the bypass handler and the software channel input interface.

There will be one decoder and buffer for each of the four input links and its function is basically to receive the header, decode it and store the incoming message in the operating system buffer.

The four bypass handlers will be activated by the respective header decoders, upon arrival of a message to be bypassed, and then, they will issue a request to the output handler, using a special soft channel, which will be uniquely identified by the link from which the message is coming and by the link through which the message is going to be forwarded, to get to its final destination. Once the output handler accepts its request, the bypass handler will release the decoder to go ahead with the retransmission of the message by the desired link.

The software channel input interface will be activated just when the message is for that transputer, and it will perform an additional check to see if the message is addressed to the Screen channel, in which case it will request permission to the screen handler to use its controlled resource. However, in both cases it will send the releasing order back to the decoder, which will send the message either to the screen, or to the appropriate channel in some waiting process.

2. **Output Handler**

This module is responsible for enforcing mutual exclusion in the four output links. Basically it will handle two kinds of messages, the outgoing ones which are generated by internal processes, and the bypassing ones which are coming from external sources, and just want to use that transputer, as a retransmission station.

This module is always listening to all possible channels, in such a way that any output request will be accepted almost immediately. Once a request for output in some specific link is accepted, that means that the requestor can go ahead with the transmission through that link, with the guarantee that no collisions will happen. As you can see it acts much like an air controller in an airport with four parallel runways, where besides ensuring mutual exclusion to each of the runways, he will keep them working in parallel.

3. **Screen Handler**

The Screen Handler will make sure that just one process from "any transputer" is holding control of the screen port at one time. Much like the output handler, once a request is accepted, the requestor is guaranteed free usage of the resource with no interference.

It is important to mention that all the communications between modules and submodules of our operating system, are done strictly via control flags, with no data flow until the very end of the process. Another point is that for efficiency purposes, we are allowing some user accessible routines like "send" and "receive", to have direct

contact with the hardware links, but this will not cause any problems, because their execution is completely controlled by the operating system modules.

IV. OPERATING SYSTEM IMPLEMENTATION

We will discuss in this Chapter all the steps and peculiarities of our implementation,[11] as shown by the source code contained in the Appendix D.

A. INPUT HANDLER

The general structure of this module is presented in Figure 4.1, where we can see that it is all the time listening to the four input links in parallel, and as soon as we have any incoming message, it will be readily consumed by the proper link.

```
PROC input.handler =
  ... variable and constant declarations
  SEQ
    ... initializing the buffers
    PAR
      WHILE TRUE
        ... listen to link0
      WHILE TRUE
        ... listen to link1
      WHILE TRUE
        ... listen to link2
      WHILE TRUE
        ... listen to link3
```

Figure 4.1 A General View of the Input Handler.

Hereafter, we recommend you to follow closely the source code contained in the Figure 4.2 for a better understanding of the program. After receiving the header with the BYTE.SLICE.INPUT built-in procedure, we start decoding the block size. You should ask why to decode the block size right away, even before knowing if that message is going or not to be bypassed, but as you will notice later, even for bypassing the block size will be required.

Once we have stored the message in the buffer of the respective link, which is maintained by the Operating System, we can proceed with the decoding. The buffer size is a very important issue, and it should be adjusted to the lengthiest message

[11] At this point is highly recommended that the user have already been exposed to the OCCAM1 programming language and to the Transputer Development System for the VAX/VMS.

expected to travel in the network. This adjustment is carried out by changing the value of the constant "max.block.size", located in the very top fold named "Operating System Global Declarations" (see Appendix D). The reason for that is because the compiler needs to allocate memory in advance for those buffers, and remember that we have one buffer per input channel.

Since we have to spend an appreciable time initializing the buffers,[12] and also because we have very strong memory limitations in the B001 board (64K), it is a wise idea to use strictly the necessary buffer size. As we will see in Chapter VII, this buffer size can be modified by changing a constant value called "max.block.size".

If the message is not for this transputer we calculate the output link by looking up in the routing table for that transputer. This table must be provided for each transputer in the network during the configuration phase, it will be covered more in detail at the end of this Chapter. But for the time being, it is sufficient to know that its indexes are the destination transputers id numbers, and the correspondent values represent the output channels to be used in order to reach those transputers. Therefore, the only valid values in the table are 4, 5, 6 or 7.

So far we haven't done anything fancy, but now comes the subtle point, I may say the most important and nice concept of the whole system. As you can see, when we send a flag to the output handler, requesting a "green sign" to go ahead with the retransmission of the header and the message, the soft channel used to send the flag must carry the necessary information, in order for the output handler to be able to recognize specifically, who is requesting permission, and which output link that request is for. The reason it must keep track of information like this is because many different users might be requesting permission to output through the same link. Now comes the question: how can we pass this information to the output handler without having to send extra bytes of information, and at the same time keeping this switching of processes as efficient as we can ? The answer we came up with was to use an array of channels whose indexes obeyed a special law of formation.

If you take a close look at the code, you will see that the channel indexes will carry all the needed information, in other words, who is requesting and what is being requested. So that, channels 04, 05, 06 and 07 will be used by any message received through link 0 that wants to be retransmitted by links 4, 5, 6 or 7 respectively.

---

[12]Although this is not a required step, it is believed that in not doing it, we may have some strange results, due to some problems in the code generation of the OCCAM1 compiler for the VAX.

```
WHILE TRUE
  -- listen to link1
  SEQ
    -- receiving the header
    BYTE.SLICE.INPUT (link1,header1,1,header.size)
    -- decoding the block size
    block.size1[0] := ((256 * header1 [BYTE 1]) + header1 [BYTE 2])
    -- buffering the message
    BYTE.SLICE.INPUT (link1,buffer.in1,1,block.size1[0])
    IF

      -- the message is to be bypassed
      header1.[BYTE 4] <> this.transputer
        SEQ
          -- finding the best link to output that message
          out1 := route.table [header1 [BYTE 4]]
          -- outputing to the required link
          BYTE.SLICE.OUTPUT(chan[10+out1],header1,3,1) --- start flag
                                                       --- thru chan 14,
          IF                                           --- 15,16 or 17
            out1 = 4
              SEQ
                BYTE.SLICE.OUTPUT (link4,header1,1,header.size)
                BYTE.SLICE.OUTPUT (link4,buffer.in1,1,block.size1[0])
            out1 = 5
              SEQ
                BYTE.SLICE.OUTPUT (link5,header1,1,header.size)
                BYTE.SLICE.OUTPUT (link5,buffer.in1,1,block.size1[0])
            out1 = 6
              SEQ
                BYTE.SLICE.OUTPUT (link6,header1,1,header.size)
                BYTE.SLICE.OUTPUT (link6,buffer.in1,1,block.size1[0])
            out1 = 7
              SEQ
                BYTE.SLICE.OUTPUT (link7,header1,1,header.size)
                BYTE.SLICE.OUTPUT (link7,buffer.in1,1,block.size1[0])
          BYTE.SLICE.OUTPUT(chan[10+out1],header1,3,1)     --- end flag

      -- the message is for this transputer
      header1 [BYTE 4] = this.transputer
        SEQ
          IF
            header1 [BYTE 3] <> scrn        --- if channel.id <> 40
              SEQ
                -- passing the size of the message (block.size1[0])
                WORD.SLICE.OUTPUT (chan[header1 [BYTE 3]],block.size1,0,1)
                -- passing the message itself
                BYTE.SLICE.OUTPUT (chan[header1 [BYTE 3]],buffer.in1,1,
                                   block.size1[0])
            TRUE                          --- if channel.id = 40 = Screen
              SEQ
                -- I'm ready
                BYTE.SLICE.OUTPUT (screen[1],header1,3,1)
                -- output to the screen
                send.string (Screen, buffer.in1, 1, block.size1[0])
                new.line(1)
                -- I'm done
                BYTE.SLICE.OUTPUT (screen[1],header1,3,1)
```

Figure 4.2 Input Handler Source Code (Partial).

Likewise, channels 14, 15, 16 and 17 will be related to link 1, channels 24, 25, 26 and 27 to link 2, and finally channels 34, 35, 36 and 37 to link 3. Therefore, these will be operating system reserved channels and must not be used as user channels. Remember, there will be no checking for this error, so that if you use these channels inside your program it will very likely deadlock the system.

When the retransmission of the message is finished, the requestor sends another flag to the output handler, to let it know that the output link may be freed or used by the next in the queue.

On the other hand, if the message is for this transputer, then a further check will be made to see if the message is to be sent to the screen or to some internal user process. This check is needed because in our implementation a message sent to the screen does not require a receiving process in the root transputer, whereas in the other case it is mandatory. Obviously this step will be carried out just in the root transputer, and this will be the basic difference between the operating system for the root and for the other transputers, since the root transputer is the only one to have a port attached to a terminal.

When the message is for some user process, the block size information is still needed, since the user process must know what is the length of the message it is about to receive, otherwise it will have no way to know when to stop receiving. This additional overhead arises from the fact that we are allowing variable length messages.

The screen channel is defined as channel "scrn" or "40", and it is another operating system reserved channel. Once we receive any message addressed to it, we will need to request permission to the screen handler, as we have done previously with the output handler, to go ahead and send it to the terminal.

B. OUTPUT HANDLER

Although the Output Handler looks very simple, it performs a very complicated task which is to assure mutual exclusion for each one of the output links.

As can be seen in Figure 4.3, all the channels with termination 4 will be polled through the first OCCAM alternate construct (ALT) to check if there is anyone requesting access to link 4. Similar action is being held for each of the others output links, with all of this being done in parallel. If there is any request for output through the hardware links, it will accept the request and will lock up that link until the user tells him that he is done. The main issue here is that the termination of the soft channel id, determines which link that channel wants to use as output.

```
PROC output.handler =
  -- local variable declarations
  VAR flag4 [BYTE 2]:
  VAR flag5 [BYTE 2]:
  VAR flag6 [BYTE 2]:
  VAR flag7 [BYTE 2]:


  PAR
    WHILE TRUE
      ALT i = [0 FOR max.io.channels]
        chan [(10*i) +4] ? flag4 [BYTE 0]       --- for link4
          BYTE.SLICE.INPUT (chan [(10*i) +4],flag4,0,1)
    WHILE TRUE
      ALT j = [0 FOR max.io.channels]
        chan [(10*j) +5] ? flag5 [BYTE 0]       --- for link5
          BYTE.SLICE.INPUT (chan [(10*j) +5],flag5,0,1)
    WHILE TRUE
      ALT k = [0 FOR max.io.channels]
        chan [(10*k) +6] ? flag6 [BYTE 0]       --- for link6
          BYTE.SLICE.INPUT (chan [(10*k) +6],flag6,0,1)
    WHILE TRUE
      ALT l = [0 FOR max.io.channels]
        chan [(10*l) +7] ? flag7 [BYTE 0]       --- for link7
          BYTE.SLICE.INPUT (chan [(10*l) +7],flag7,0,1):
```

Figure 4.3 The Output Handler.

Although it is not clear at a first glance, due to the way that the replicator ALT is implemented in OCCAM1, we are having a sequential, rather than parallel, output through the links 4, 5, 6 and 7. Let's suppose we have somewhere in time the following channels requesting output in parallel: chan[17], chan[4], chan[35], chan[54], chan[76], chan[84], chan[107] and chan[66] (see Figure 4.4). We should be able to realize by now that the first three channels are reserved channels and carry some messages which must be bypassed through links 7, 4 and 5 respectively. On the other hand, the remaining five channels are being used by some internal user processes running in that transputer, which want to use, respectively, links 4, 6, 4, 7 and 6 for output. What we should expect by looking at our implementation of the output handler (see Figure 4.3), would be to have the following sequence of transmissions,[13] as depicted in the Figure 4.4, but what actually happened was a sequential transmission in the following order: chan[17], chan[4], chan[54], chan[84], chan[35], chan[66], chan[76] and finally chan[107] (see Figure 4.5).

[13]We have assumed the same length for all the messages just for convenience.

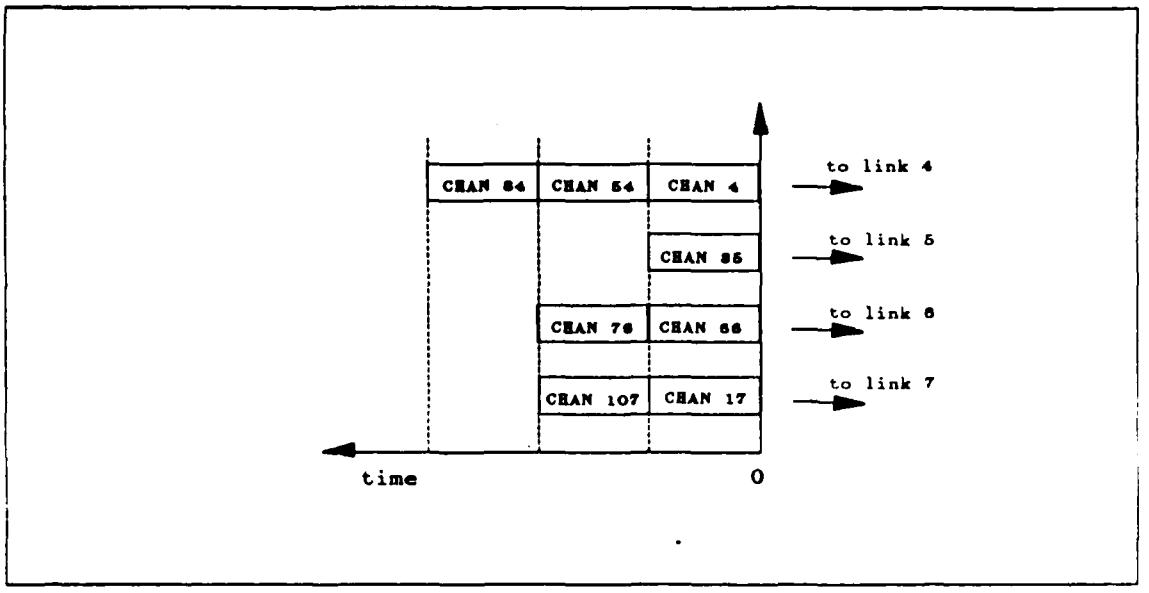


Figure 4.4 The Expected Behaviour.

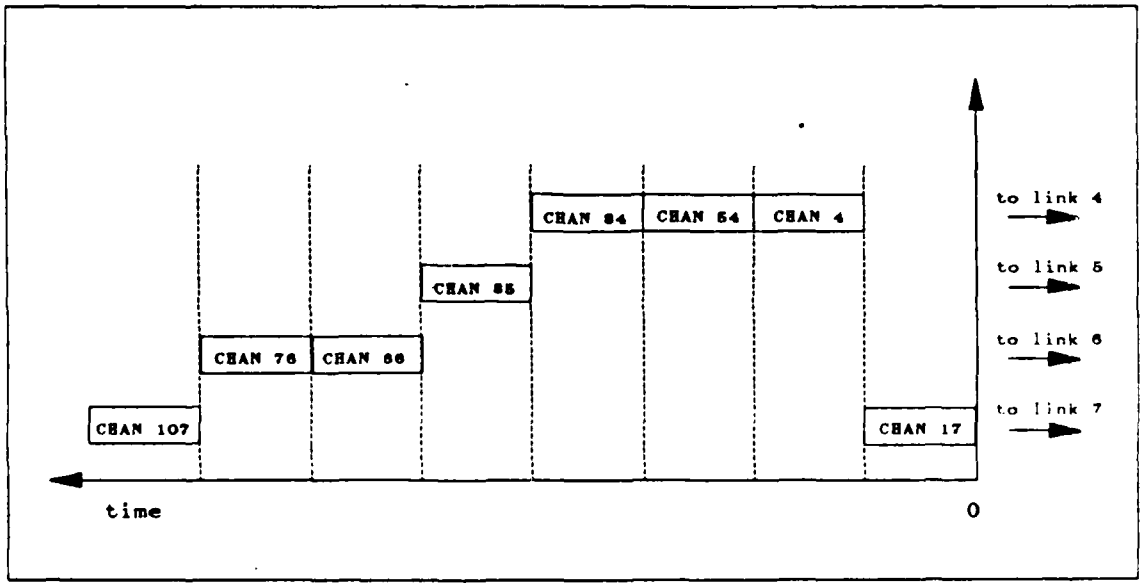


Figure 4.5 The Actual Behaviour.

Based on the previous results, we couldn't come up with a reasonable explanation, other than some problem in the code generation of the OCCAM1 compiler for the VAX/VMS.

After many different trials we have finally got a way to implement it truly in parallel, and it came up with the odd structure presented in Figure 4.6.

```
WHILE TRUE
  SEQ
    going1 := TRUE
    going2 := TRUE
    going3 := TRUE
    going4 := TRUE
    PAR

      WHILE going1
        ALT
          ALT j = [0 FOR max.actual.channels]
            chan [(10*j)+4] ? flag1
              BYTE.SLICE.OUTPUT(chan[(10*j)+4],flag1,1,1)
          SKIP
        going1 := FALSE

      WHILE going2
        ALT
          ALT k = [0 FOR max.actual.channels]
            chan [(10*k)+5] ? flag2
              BYTE.SLICE.OUTPUT(chan[(10*k)+5],flag2,1,1)
          SKIP
        going2 := FALSE

      WHILE going3
        ALT
          ALT l = [0 FOR max.actual.channels]
            chan [(10*l)+5] ? flag3
              BYTE.SLICE.OUTPUT(chan[(10*l)+6],flag3,1,1)
          SKIP
        going3 := FALSE

      WHILE going4
        ALT
          ALT m = [0 FOR max.actual.channels]
            chan [(10*m)+7] ? flag4
              BYTE.SLICE.OUTPUT(chan[(10*m)+7],flag4,1,1)
          SKIP
        going4 := FALSE
```

Figure 4.6 The Parallel Solution.

However, this was the only way we found to trick the compiler. With this structure we had the expected parallel output, in other words, each link transmitting in parallel, exactly like depicted in Figure 4.4.

If the reader looks at the final implementation of the Output Handler, he will notice that we have used the first structure, rather than the second one. There are some reasons for that, the first one is because we have assumed that it will be very unlikely to have such a situation where all the channels will be ready exactly at the same time

and furthermore, we are talking about more than one message ready for each of the links at the same time, what you should agree that will be quite unusual, but nevertheless, the most important reason was that after evaluating both structures, we have ended up with a very big difference in execution time towards the first one, being more specific, it was in average about 5.6 times faster than the second one ! Actually, this could be partially expected just by looking at the overhead imposed by the second structure.

Now you can realize why we have chosen an array of channels, instead of using generic names for those channels which communicate with external transputers. Stop and think how difficult and cumbersome it would be, to make an alternate construct with generic names for their guards.[14]

C. SCREEN HANDLER

The idea behind this procedure is exactly the same as the output handler. The main difference is that it will take care just of one channel, the Screen.

If any other transputer wants to output to the screen, the only thing it must do is to use the standard "send" routine, which will be covered later in the Section dealing with the Library Routines, and send any message he wants through the Operating System reserved channel 40, also defined as "scrn".

Once this has been done, the message will arrive at the input handler of the root transputer, and after being decoded it will end up in requesting permission to the screen handler to output the message to the screen. If you look carefully in the input handler code, the software channels screen[0], screen[1], screen[2] and screen[3] are directly related to messages coming from external sources. If some process in the root transputer wants to output something to the screen, it will use additional software channels allocated for it in advance. In the actual implementation we have reserved just two more screen channels for the root transputer, screen[4] and screen[5], but this is a matter of just changing the constant "max.screen.channels", which is located in the "Operating System Global Declarations" fold, and you will be able to have as many as you need for your application.

This feature actually improves the capability of the programmer, since he had prior to this implementation just one possible channel for writing to the screen, and now we can have as many as we want. Obviously, due to physical limitations (we have

[14] *Guard* is the name given for the channels which are being polled for input by the alternate construct.

just one port to the monitor), we will have a sequential output, but in doing so we are taking from the user the need to worry about this matter, in other words, we are providing to the user a higher level of abstraction.

However, the beauty of all this about the screen, is that we have now the possibility of debugging remote transputers, what we didnt't have before. Now we can even trace the execution of all our processes running in the entire network, by just sending a flag to the screen when entering in some procedure and when exiting it. Of course it won't be in real time, but at least we will be able to have a precise idea of the entire flow of control in the network.

If we have the case where four transputers may try to output to the screen at the same time, we won't know which is which, so that we must send a unique message which characterizes the transputer it is coming from. This little problem could be very easily solved by inserting the origin transputer id in the message header, but for efficiency purposes we decided not to implement it.

Finally, you might have already noticed that the output to the screen from remote transputers, is the only send operation which does not require any other receive operation in the root transputer. This is a basic point, since for every send operation ought to have a receive operation for the same channel id somewhere in the network, otherwise the system will deadlock.

D. THE ROUTING TABLE

The routing table is the instrument that will provide to the operating system, the necessary information regarding the routing of messages. For example, if we receive a message which needs to be bypassed, the O.S. based on the destination transputer id for that message, will look up in the table and see which is the recommended link to output that message. Once this has been determined, it will follow the standard steps to input or output a message, as already discussed in previous Sections. Similar action will be taken for internal messages, which want to be forwarded.

In the first implementation of our system, we put the routing table, as well as the transputer id number, as global variables inside each of the SC PROCs to be downloaded to the network. Although this way is much simpler to implement, it presents a very serious limitation, and that happens when we want to load basically the same SC PROC in several transputers, just differing by its id number and by its routing table. If that was the case we would have to make as many as needed different SCs, which is completely wasteful. So, we decided to pass the transputer id number and the

routing table as constant parameters in the configuration, but since OCCAM1 does not support tables as parameters in the configuration,[15] the only solution was to pass value by value, and if we follow the code we will see a fold in the top of the main program of the operating system, where the entire routing table is received.

Another point to mention is that our routing table was limited to 18 entries because we have just 18 transputers in our Lab. But if for some reason we need to change it, we must carry out the following steps:

1. Change the route.table declaration, which is located in the "Operating System Global Declarations" fold, in each of the files ROOT_OS.TDS and REMOTE_OS.TDS;
2. Add as many new parameters as needed to each of the SC PROCs to be downloaded in the different transputers;
3. Go to the main body of the PROC "operating.system", where the routing table is actually received, and assign the new parameters from the previous step, to some new indexes of the routing table. Try to be consistent with the assignments which are already in there;
4. In the configuration Section, add the new values to the parameter list of each of the SC PROCs, much like you did in step 2;

After building the routing table, the user must check it for the non existence of cycles in it. If that happens, it will be very likely that some of the messages will never arrive in their final destination, constituting a very difficult problem to isolate. Hence, it is strongly recommended to perform this test, before using a new routing table.

As the reader can notice, our routing table is nothing else than a graph, so that if the table is very large, it is recommended to make a little program to detect cycles in it. There are many algorithms to detect cycles in a graph, which might be found in any book covering Graphs. However, if the table is small, it is quite simpler to check it by hand.

Suppose we are given a transputer network and a routing table, as specified in Figure 4.7. The routing table specify the output links, which must be used by the origin transputers, when they desire to send some message to the destination transputers.

We suggest the following algorithm:

1. Select a column, in other words, a destination transputer;
2. Select an origin transputer, one at a time, and use the output link listed in the table, to find by looking at the network, where the message will be directed to;

---

[15]It is believed that the new beta release version of OCCAM2 for the VAX/VMS will support tables as parameters in the configuration.

3. If the transputer obtained from Step 2, is not the destination transputer, jump back to Step 2 and use the latter one as the new origin transputer, otherwise, proceed to Step 4;
4. If you have got a cycle when executing Steps 2 and 3 stop, modify the routing table, and start all over again, otherwise, jump back to Step 1 and select another column, until all columns have been selected.

E. OPERATING SYSTEM LIBRARY ROUTINES

What do these libraries contain ? Is it required to put them in any TDS program? As explained in Chapter II, the answer is "no". They are not required to be inside a file, a fold or whatever, but this was the way we found most attractive, logical and easy, to handle the increasing number of new procedures, which were created along our research. The basic idea is to update the library whenever someone in the research group, has made a routine which might be useful in the future. However, it will be useless if it is not very well documented, tested and validated for all expected inputs.

Following this line of thought we have built basically two libraries, the first one for the root transputer, and the second one for remote transputers. Both are completely described within Appendices D and E, which present the source code for the root and remote operating systems.

Although they have almost the same routines, they have quite a few differences in their implementations, as we should expect. However, it is not our intention to provide a deep explanation of all the code contained in those libraries, but the interested reader is welcome to go into the source code, which we have tried to document as well as possible.

The only routines which we will cover in detail are the "send" and "receive" routines, which are the basic interface to the user, and also the most important procedures to handle communications in the network.

1. The Send Routine

When we want to send a message to an external transputer, this is the right procedure to call. It is one of the two only routines, which can access the modules of the operating system directly. It is also, as we will see later, the only routine that can output directly to the hardware channels, but obviously under control of the operating system. The fact that we are allowing a user accessible routine to talk directly with the hardware output link, may cause the unpleasant feeling that we are not following our previous abstraction model for our system, but that is not correct, since the operating system is still with the control over that link. Furthermore, we have tried other ways and this was by far the most efficient one.

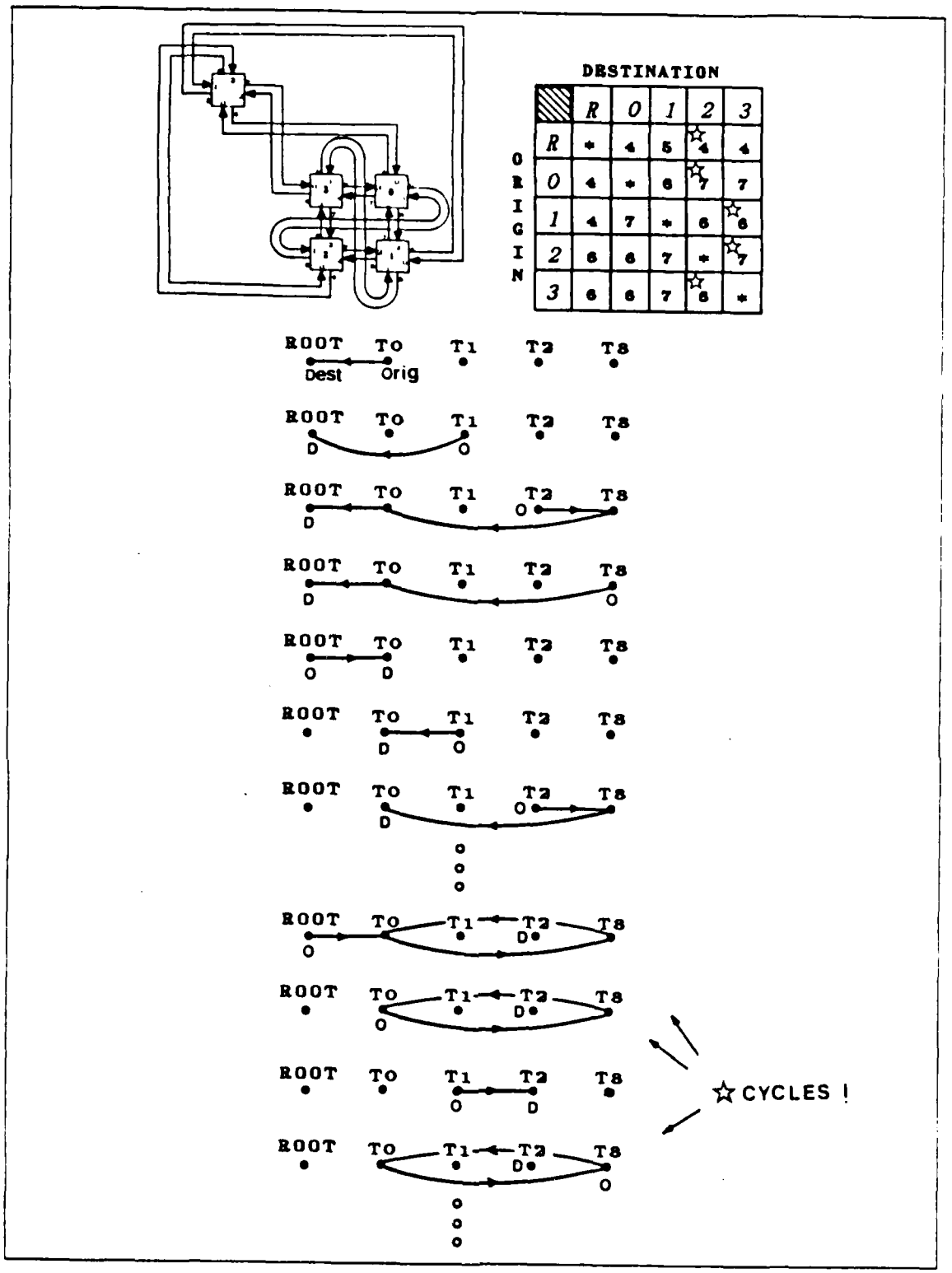


Figure 4.7 Checking the Routing Table for Cycles.

It is inside this procedure where the header is constructed, using the information provided by the user. Another important point to raise is that for any "send" we must have a "receive" for that same channel id, exactly at the destination transputer for that send. The only exception is in the case of the channel "40" or "scrn", where we don't need to have a receiving process for the send.

The following parameters must be passed to this procedure:

- The channel id which is an integer value multiple of 10, in the range of 40 up to 240. Remember that channels 0 up to 37 are operating system reserved channels and cannot be used inside our program. The only reserved channel that we can make use of, is the "40", which is uniquely assigned to the Screen channel, but even though, we cannot use it to send messages to transputers others than the root;
- The destination transputer id is a unique integer value which characterizes a transputer. The value assigned to it must be inside the range of our routing table, in other words, if we have a routing table with 18 entries, the id numbers must be between 0 and 17 including both endpoints;
- The start byte is the position of the first byte in the array that we want to send. Obviously it cannot be negative. It is important to remember that an array in OCCAM always start from byte 0, so that "start.byte" equal to 0 is a valid entry. However, always keep in mind what we really want to send;
- The size is the number of bytes to be transmitted. If we want to send the message from some specific start byte up to the end of the array, we don't have to make any calculations, just put "0" as the size value. However, in using this latter approach, it is mandatory that in the byte 0 of this array we have its size information, as it is usually done in OCCAM1.

send (VALUE chan.id,dest.transp,message[ ],start.byte,size)

USAGE: send (60,15,2,7)

send (scrn,1,5,0)

2. **The Receive Routine**

This procedure is the second and last routine which have direct contact with operating system modules, specifically the Input Handler. It basically receives the message that was sent by the correspondent "send" routine, and since we are allowing variable message size, it returns to the user, as an output parameter, the length of the received message. It requires the following parameters:

- The channel id which is an integer value, multiple of 10, in the range of 50 up to 240. Much likely the "send", we cannot use the operating system reserved channels, and with the "receive", not even the "scrn" channel or "40" is accepted;
- This parameter is an output parameter of the type array, and contains the variable to hold the incoming message, which must be declared as an array of the same type of the original message;
- The last parameter is also an output parameter and provides the user with the length of the message just received. It is included as a parameter, because we might have a case where the program must take an action based on the length of the message, but in most of the cases it can be disregarded. It must be declared as an array of integers with size 1.

receive (VALUE channel.id, VAR message[ ],message.length[ ])

USAGE: receive (60,message.in.size)

3. **The Root Library (ROOT_LIB.TDS)**

This Library is intended to be used under the Operating System in the root transputer. It contains basically many I/O routines with various formatting capabilities, some conversion routines to change the representation of some data types, and some utilities which will help you in getting the real time inside some process, to calculate the transfer rate in KBits/sec of some link, to dump parts of memory for debugging purposes, etc....

Unfortunately, the file system interface with the VAX/VMS is not supported by the OPS Kernel in OCCAM1, and although we have put some effort in solving this problem, we didn't have the sufficient time to get a successful result. This issue will be discussed later in the Section covering follow-on work, in Chapter VII.

The Appendix D will present the operating system for the root transputer, and in there, you will be able to find the filed fold ROOT_LIB.TDS, with all the routines in it.

4. **The Remote Library (REMOTE_LIB.TDS)**

The basic difference between both libraries is that in the first one, we can send anything directly to the screen, after receiving the consent of the Screen Handler, whereas in the second, we must always use the procedure "send", by the special channel "40" or "scrn". Another difference is that in the second library, we don't have the

PROC rem.write.string, simply because the PROC send performs the same function, with even some more enhancements. Finally, we don't have the PROCs rem.read.string and rem.read.number, which use the channel Keyboard for input. They were not implemented because of time constraints in our schedule, but we will give a brief suggestion for their implementation, in the Section covering "follow-on work".

Similarly, the REMOTE_OS.TDS which is presented in Appendix E, will contain the REMOTE_LIB.TDS as a filed fold.

V. EVALUATION OF THE OPERATING SYSTEM

A. INTRODUCTION

This Chapter will be devoted to one of the main issues, when dealing with programs which are supposed to run under a real time environment, and that is its "performance evaluation".

The evaluation will be basically software oriented, in the sense that no hardware measurements will be made. Strictly speaking, it will consist in running a special program, where all links will be exercised in transmitting messages of different sizes, and where the parallel operation of the links will be stressed as well.

B. A BRIEF DESCRIPTION OF THE EVALUATION

The configuration on which we are going to evaluate our operating system, is composed of a root transputer directly connected to four others transputers, as presented in Figure 5.1. The program used for the evaluation is a modification of the evaluation program made by Vanni J.F. in his thesis [Ref. 9], where a complete evaluation of the Transputer and its communications links is presented. We will be using even the same configuration, in such a way, that our final results for the link transfer rates, when using the Operating System, can be fully compared with his results.

Basically, what his program does is to send successively to one transputer, then to two, three, four and finally to all transputers in parallel, messages with varying sizes, starting from 1 byte up to 10000 bytes.[16] After receiving these messages the remote transputers will echo them back, also in a parallel fashion, and the root transputer after a very careful and precise timing process, will time the entire transfer process and display the transfer rate in KBits/sec.

The structure of this evaluation program can be better understood, if stated as a sequence of steps, where each complete sequence will be applied, consecutively, for each message size. We will also present in Figure 5.2, a partial view of the evaluation program, which will be running in the root transputer.

---

[16]In our modified version of his program, we will be limited to a maximum message size of 4K, due to B001 board memory limitations, which are aggravated by the fact that the Operating System must maintain a series of buffers in addition to the user declared ones.

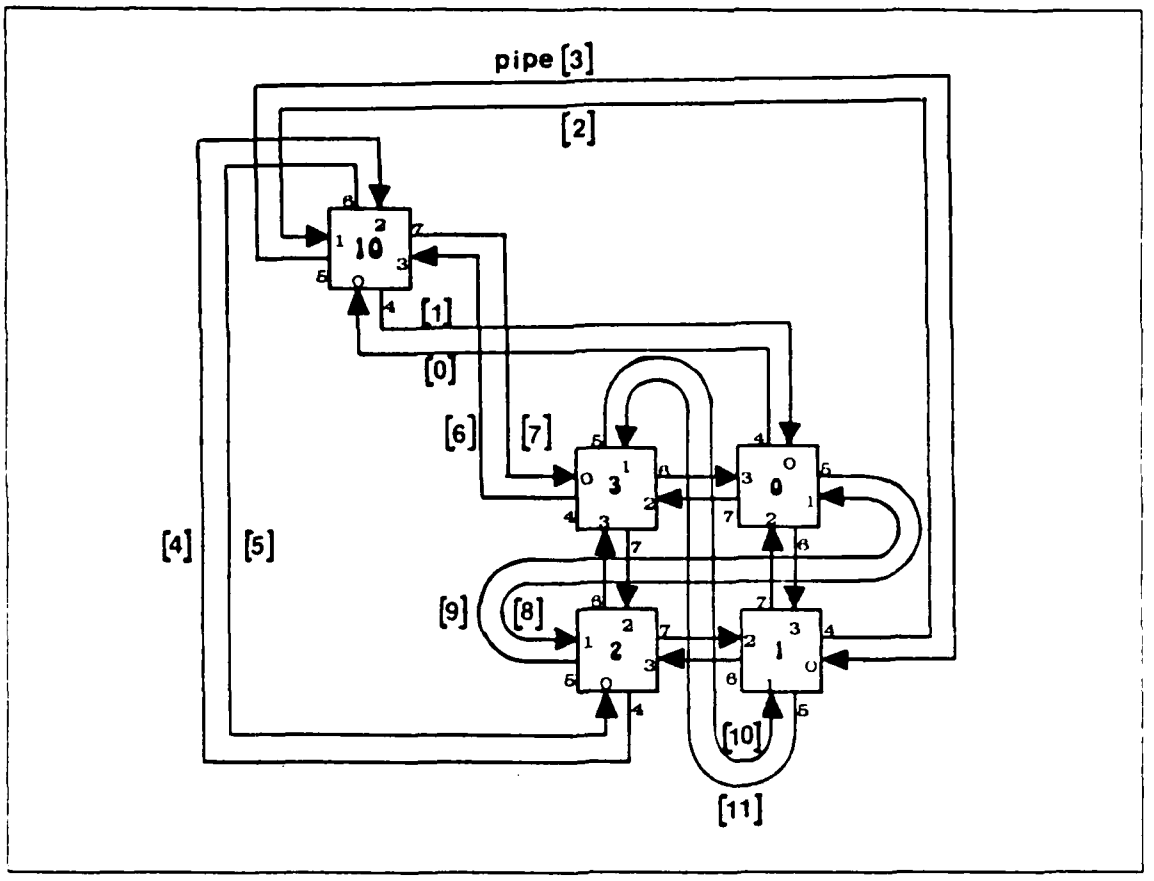


Figure 5.1 The Configuration used in the Evaluation Process.

1. A flag is sent from the root transputer to transputer #0. When the flag is accepted it will mean that the transputer #0 is ready to receive the actual message. The end of this step will determine the start of our timing process, which is carried out in the root transputer. The basic objective of this flag is to achieve the most accurate synchronization between processors as possible, since it will directly influence the precision of our results;
2. The actual message is sent, and once it is received by transputer #0, we will stop timing;
3. The transfer rate in KBits/sec is calculated and displayed as "1OUT";
4. Another flag is sent to transputer #0 for the same reasons specified in Step 1. Once it is received by the transputer #0, we will start timing in the root transputer;
5. The transputer #0 echoes back the message to the root transputer. Once the entire message is received the timing process is stopped;
6. The transfer rate is calculated and displayed as "1IN";

7. Two flags are sent in parallel to transputers #0 and #1. Upon arrival we start timing;
8. The message is sent to both transputers, in parallel, by 2 different links. Upon arrival of both messages we stop timing;
9. The transfer rate is calculated and displayed as "2OUT";
10. Two more flags are sent in parallel to transputers #0 and #1. Upon arrival we start timing;
11. The messages are echoed back by both transputers. Upon arrival of both messages in the root transputer, we stop timing;
12. The transfer rate is calculated and displayed as "2IN";
13. Three flags are sent in parallel to transputers #0, #1 and #2. Upon arrival we start timing;
14. The message is sent to the three transputers, in parallel, by 3 different links. Upon arrival of the messages we stop timing;
15. The transfer rate is calculated and displayed as "3OUT";
16. Three more flags are sent in parallel to transputers #0, #1 and #2. Upon arrival we start timing;
17. The messages are echoed back by the transputers. Upon arrival of all messages in the root transputer, we stop timing;
18. The transfer rate is calculated and displayed as "3IN";
19. Four flags are sent in parallel to transputers #0, #1, #2 and #3. Upon arrival we start timing;
20. The message is sent to the four transputers in parallel by 4 different links. Upon arrival of the messages we stop timing;
21. The transfer rate is calculated and displayed as "4OUT";
22. Four more flags are sent in parallel to transputers #0, #1, #2 and #3. Upon arrival we start timing;
23. The messages are echoed back by the transputers. Upon arrival of all messages we stop timing;
24. The transfer rate is calculated and displayed as "4IN";
25. Four flags are sent in parallel to transputers #0, #1, #2 and #3. Upon arrival we start timing;
26. The message is sent to the four transputers, in parallel, by 4 different links, and they will echo them back immediately. Upon return of the four messages at the root transputer, we will stop timing. This step is carried out just to check the performance when all 8 channels (2 per link), are working in parallel;
27. The transfer rate is calculated and displayed as "4INOUT";

```
PROC transfer =
  SEQ
    SEQ i = [0 FOR nr.of.sizes]
      SEQ
        block.size := sizetable[i]
        -- output to one channel
        actual.rate := 0
        SEQ j = [1 FOR repetition]
          SEQ
            send (90,0,"a ",1,1)
            TIME ? time0[0]
            send (90,0,buffer0,1,block.size)
            TIME ? time1[0]
            transfer.rate(time0[0],time1[0],1,block.size,rate)
            actual.rate := ((actual.rate * (j-1)) + rate)/j
        SKIP

                    o
                    o
                    o

        -- input from one channel
        -- output to two channels
        -- input from two channels
        -- output to three channels
        -- input from three channels
        -- output to four channels
        -- input from four channels

                    o
                    o
                    o

        -- all output and input in parallel
        actual.rate := 0
        SEQ j = [1 FOR repetition]
          SEQ
            PAR
              send (90,0,"a ",1,1)
              send (100,1,"a ",1,1)
              send (110,2,"a ",1,1)
              send (120,3,"a ",1,1)
            TIME ? time0[0]
            PAR
              send(90,0,buffer0,1,block.size)
              send(100,1,buffer1,1,block.size)
              send(110,2,buffer2,1,block.size)
              send(120,3,buffer3,1,block.size)
              receive(50,buffer0,dummy0)
              receive(60,buffer1,dummy1)
              receive(70,buffer2,dummy2)
              receive(80,buffer3,dummy3)
            TIME ? time1[0]
            transfer.rate(time0[0],time1[0],1,block.size,rate)
            actual.rate := ((actual.rate * (j-1)) + rate)/j
        SKIP

    SKIP
```

Figure 5.2 The Transfer Program in the Root Transputer (Partial).

Although this synchronization procedure through the use of flags is very accurate when using direct connections of links, that is not quite true when using the Operating System, and that is because we will be sending the flags, which are our time reference, through the same Operating System we want to evaluate. This latter uncertainty will have the effect of slightly decrease the actual measured transfer rate. Nevertheless, this is still the best way we found to get a sufficiently accurate result.

Another point to mention, is that there will be a constant called "repetition", which specifies the number of times to perform each of the above steps, in order to calculate an average value for the transfer rates.

C. EXPERIMENTAL RESULTS

1. Evaluating Direct Communications

The tables must be interpreted according to the sequence of steps presented in Section A of this same Chapter. For example, if we look at Tables 1 and 2, we will see that when using one channel to output a message of 2048 bytes, we obtain a transfer rate of 3423 KBits/sec with the operating system, and a rate of 3658 KBits sec without it, so that we can say we have lost 6% in the speed for this specific message size. Of course, this percentage tends to increase as we decrease the message size. In the most unfavorable situation, we will be trying to output and to receive a message of one byte, through the 8 channels, in which case we will have speed losses of the order of 96%. However, it is important to notice that we are comparing the operating system with the fastest available construct, which is the "BYTE.SLICE" [Ref. 9].

As the reader can notice, the rates for the "INs" are somewhat higher than for the "OUTs", but as the message size increases, they tend to equalize. We believe that the reason for that, is intimately related with the time spent by the synchronization flag, to pass through the operating system, and aggravated by the fact that the root transputer (T414-12) is a slower machine than the remote ones (T414-15). This extra overhead is expected to present a decreasing relative contribution, as the total transfer time goes up, which is a direct consequence of the message size. As one can see, this last assumption agrees with the fact that they tend to equalize.

Our next step in the evaluation process was towards the use of high priority for the operating system. As can be seen in Table 3, the rates for the "OUTs" have presented a modest increase, whereas the "INs" had a very substantial increase. However, as the reader can notice, the rates for "3IN" and "4IN" are not consistent, since we have smaller messages with higher rates than bigger ones, which is not correct.

We have no explanation for such a behavior, but we suppose that it might have some relation with the fact, that the user process, which has low priority, is accessing the send and the receive routines, which are high priority operating system routines. This, somehow, might be causing some sort of problem for the scheduler. However, although it was not checked, we believe that the data integrity was maintained, otherwise a deadlock would have occurred.

After analyzing the experimental results, seems to us that the assignment of high priority to the operating system is the right way to go, despite of the problem presented in the previous paragraph. However, it is suggested additional investigation, before using the PRI PAR construct in the operating system.

As a summary, we will present in Figure 5.3, a comparison between the best and the worst cases from Tables 1, 2 and 3.

TABLE 1

TRANSFER RATES WITHOUT THE OPERATING SYSTEM
BETWEEN ADJACENT TRANSPUTERS (KBITS/SEC)

BYTES	1OUT	1IN	2OUT	2IN	3OUT	3IN	4OUT	4IN	4INOUT
1	625	616	250	250	200	198	161	161	98
2	1217	1237	500	500	400	400	325	333	196
4	1531	2130	779	1000	648	788	650	646	384
8	2183	2811	1570	1582	1311	1301	1085	1096	690
16	2758	2924	2101	2222	1948	1919	1702	1694	1255
32	3224	3246	2589	2800	2482	2544	2330	2398	1835
64	3427	3646	3116	3226	2942	3048	2817	2954	2462
128	3543	3644	3332	3497	3265	3390	3187	3320	2945
256	3605	3741	3496	3656	3444	3596	3398	3558	3231
512	3635	3778	3578	3733	3555	3697	3509	3677	3401
1024	3650	3754	3627	3741	3604	3712	3575	3702	3512
1280	3654	3748	3640	3742	3611	3713	3587	3698	3529
2048	3658	3740	3652	3738	3621	3715	3604	3703	3549
4096	3662	3735	3663	3733	3634	3720	3618	3709	3573

2. Evaluating Multiple Path Communications

So far, we have been evaluating the operating system when working with adjacent transputers, which is not the most clever way of using it. However, when we have transputers not directly interconnected, which need to communicate among themselves, it becomes almost a must.

TABLE 2

TRANSFER RATES WITH THE OPERATING SYSTEM
BETWEEN ADJACENT TRANSPUTERS (KBITS/SEC)

BYTES	1OUT	1IN	2OUT	2IN	3OUT	3IN	4OUT	4IN	4INOUT
1	21	32	9	20	6	14	5	10	4
2	43	64	19	41	13	28	10	21	9
4	85	127	38	82	27	57	21	43	18
8	166	248	75	161	54	112	42	85	37
16	319	464	147	308	106	217	83	167	73
32	588	820	283	565	208	406	163	317	146
64	1015	1327	526	963	394	727	313	576	288
128	1596	1938	922	1510	710	1189	577	972	507
256	2225	2521	1476	2092	1188	1739	996	1480	943
512	2777	2967	2108	2599	1795	2266	1565	2003	1503
1024	3175	3256	2671	2955	2408	2673	2190	2434	2166
1280	3273	3322	2831	3038	2584	2774	2381	2545	2423
2048	3423	3420	3086	3168	2919	2930	2738	2727	2650
4096	3556	3490	3329	3267	3238	3064	3132	2885	2824

TABLE 3

TRANSFER RATES WITH THE OPERATING SYSTEM (HIGH PRI)
BETWEEN ADJACENT TRANSPUTERS (KBITS/SEC)

BYTES	1OUT	1IN	2OUT	2IN	3OUT	3IN	4OUT	4IN	4INOUT
1	23	71	11	131	7	86	5	65	5
2	46	140	23	263	15	173	11	129	10
4	92	270	47	522	31	294	23	256	21
8	180	506	93	701	62	588	47	506	43
16	342	888	182	1415	123	1141	92	987	84
32	626	1427	346	1998	239	2086	180	1860	164
64	1070	2042	633	2520	448	3498	344	3355	312
128	1661	2601	1079	2898	798	3260	630	5063	561
256	2296	3026	1671	3137	1312	3440	1074	4394	921
512	2830	3288	2300	3273	1937	3417	1665	3631	1392
1024	3205	3438	2830	3344	2538	3355	2294	3344	2183
1280	3294	3470	2967	3359	2705	3319	2479	3292	2371
2048	3437	3507	3200	3374	3001	3287	2825	3213	2732
4096	3561	3519	3424	3369	3307	3241	3196	3131	2874

In this subsection, we are going to evaluate the performance of the operating system for a multiple path communication, or sometimes referred as multiple hops communication. The tables 4, 5 and 6 present the transfer rates for 1, 2 and 3 retransmissions, which actually corresponds to 2, 3 and 4 hops, respectively. It is

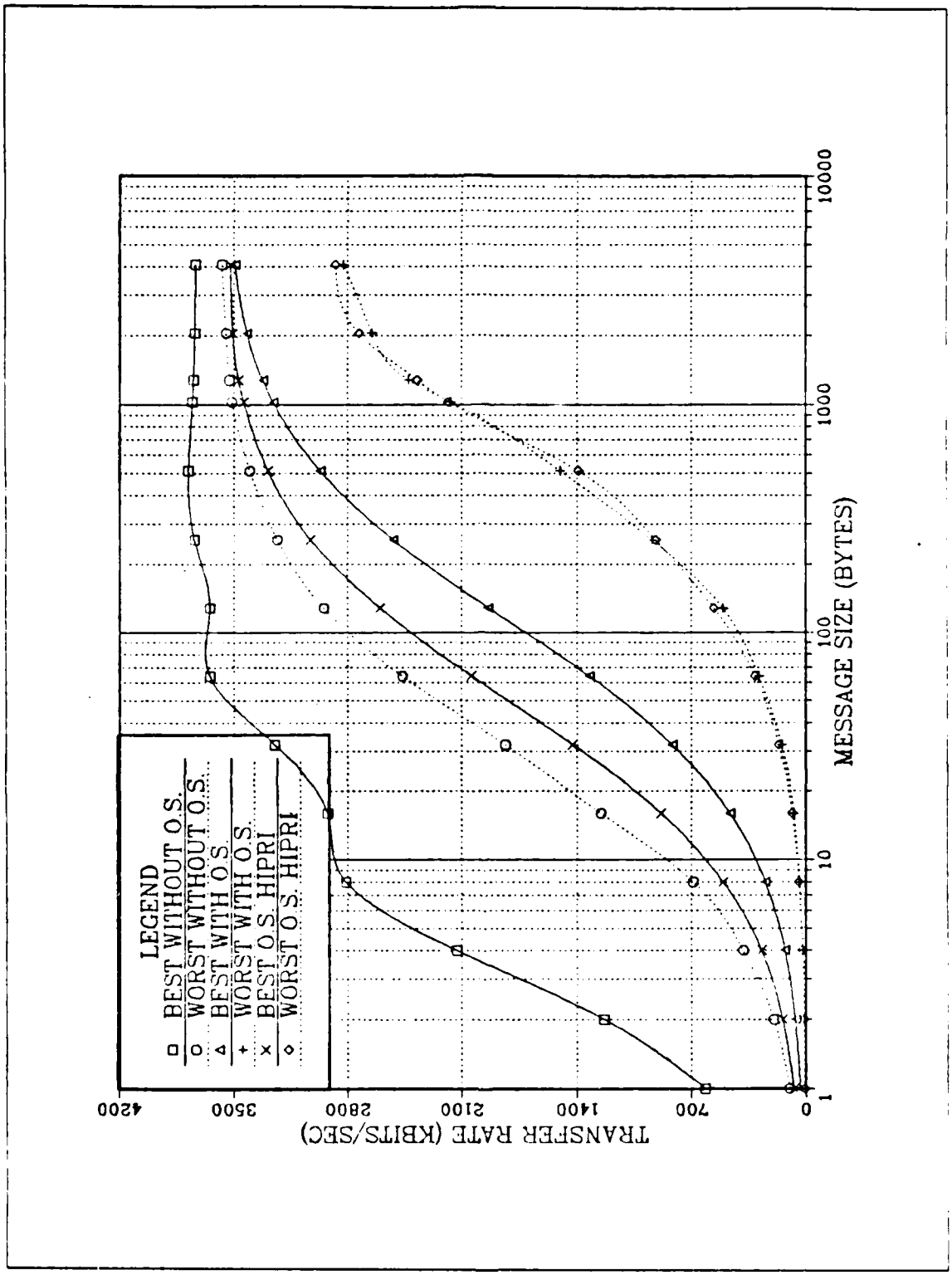


Figure 5.3 Transfer Rates with Direct Communications.

worth mentioning, that the retransmission process is not a strict key switching, where the "link transputer" would have just to connect its input to the output links directly, and that is it. Actually, each transputer in order to retransmit a message, should have to receive and store the entire incoming message into its local memory, and only then, start the retransmission process. That is basically what is going to be evaluated in this Section.

The program will be basically the same, with a slightly modification in the routing table. For the one retransmission case (2 hops), we have forced the transputer 0 to send its messages to the root transputer, via transputer 2. For the 3 hops case, via transputers 2 and 1, and finally, for the 4 hops case, via transputers 2, 1 and 3.

However, it is important to mention that the transputers which are going to be used as "links" are overloaded, since they are still sending their own messages as well. Hence, this evaluation is going to be a sort of "worst case" evaluation. It is believed, that if the other processors were not executing any processes other than the operating system, these transfer rates would have presented a substantial increase. When examining these tables, notice that the "OUTs" are about the same as in Table 2, with adjacents transputers, and that happens because the root transputer is still sending the messages directly. Therefore, only the "IN" columns will be of interest, in this step of the evaluation. See in Figure 5.4 a comparison between Tables 4, 5 and 6.

TABLE 4

TRANSFER RATES WITH THE OPERATING SYSTEM
IN 2 HOPS (KBITS/SEC)

BYTES	1OUT	1IN	2OUT	2IN	3OUT	3IN	4OUT	4IN	4INOUT
1	21	17	9	18	6	9	5	8	3
2	43	35	19	37	13	19	10	16	7
4	85	68	38	72	27	38	21	32	14
8	166	132	75	138	53	74	42	63	29
16	319	249	147	258	106	143	83	123	59
32	588	438	283	454	207	266	164	230	118
64	1013	707	526	725	391	465	313	410	234
128	1588	1018	919	1036	707	742	576	670	438
256	2216	1312	1473	1312	1185	1054	995	980	773
512	2769	1531	2103	1510	1793	1336	1564	1276	1163
1024	3158	1672	2671	1633	2406	1539	2194	1458	1357
1280	3250	1701	2826	1662	2583	1574	2383	1498	1389
2048	3399	1748	3087	1723	2907	1629	2745	1563	1450
4096	3528	1782	3327	1770	3237	1674	3136	1620	1530

TABLE 5

TRANSFER RATES WITH THE OPERATING SYSTEM IN 3 HOPS (KBITS/SEC)

BYTES	1OUT	1IN	2OUT	2IN	3OUT	3IN	4OUT	4IN	4INOUT
1	21	12	9	11	6	6	5	6	3
2	43	24	19	22	13	13	10	12	7
4	85	47	38	44	27	27	21	24	14
8	166	91	75	85	53	52	42	47	29
16	318	170	147	161	106	99	83	89	57
32	588	300	283	284	207	179	164	162	111
64	1012	484	526	464	392	301	313	277	215
128	1588	694	921	671	707	454	576	426	371
256	2216	890	1472	870	1187	610	996	583	540
512	2766	1035	2104	1020	1791	735	1565	718	681
1024	3159	1130	2672	1119	2407	821	2192	807	778
1280	3250	1150	2822	1140	2586	839	2384	825	799
2048	3399	1180	3087	1174	2905	872	2745	854	824
4096	3529	1207	3330	1203	3239	893	3133	877	851

TABLE 6

TRANSFER RATES WITH THE OPERATING SYSTEM IN 4 HOPS (KBITS/SEC)

BYTES	1OUT	1IN	2OUT	2IN	3OUT	3IN	4OUT	4IN	4INOUT
1	21	9	9	9	6	6	5	4	3
2	43	18	19	19	13	13	10	9	6
4	85	36	38	38	27	26	21	18	12
8	166	70	75	72	53	50	42	36	25
16	318	132	147	135	106	95	83	69	49
32	588	231	283	235	207	167	164	124	94
64	1011	371	526	377	392	268	313	208	172
128	1586	532	921	539	707	374	576	312	284
256	2216	678	1473	683	1186	469	995	416	392
512	2767	787	2103	790	1792	536	1564	499	483
1024	3157	855	2669	858	2405	578	2195	556	542
1280	3248	871	2823	872	2583	586	2383	567	556
2048	3396	895	3082	895	2909	601	2741	588	578
4096	3529	914	3327	914	3239	612	3134	603	591

D. EFFECT OF THE HEADER SIZE IN THE TRANSFER RATE

One of our main concerns in the design phase of the communications protocol, was regarding the size of the header. We thought that decreasing the header size by half, for example, we would almost double the performance, mainly when dealing with

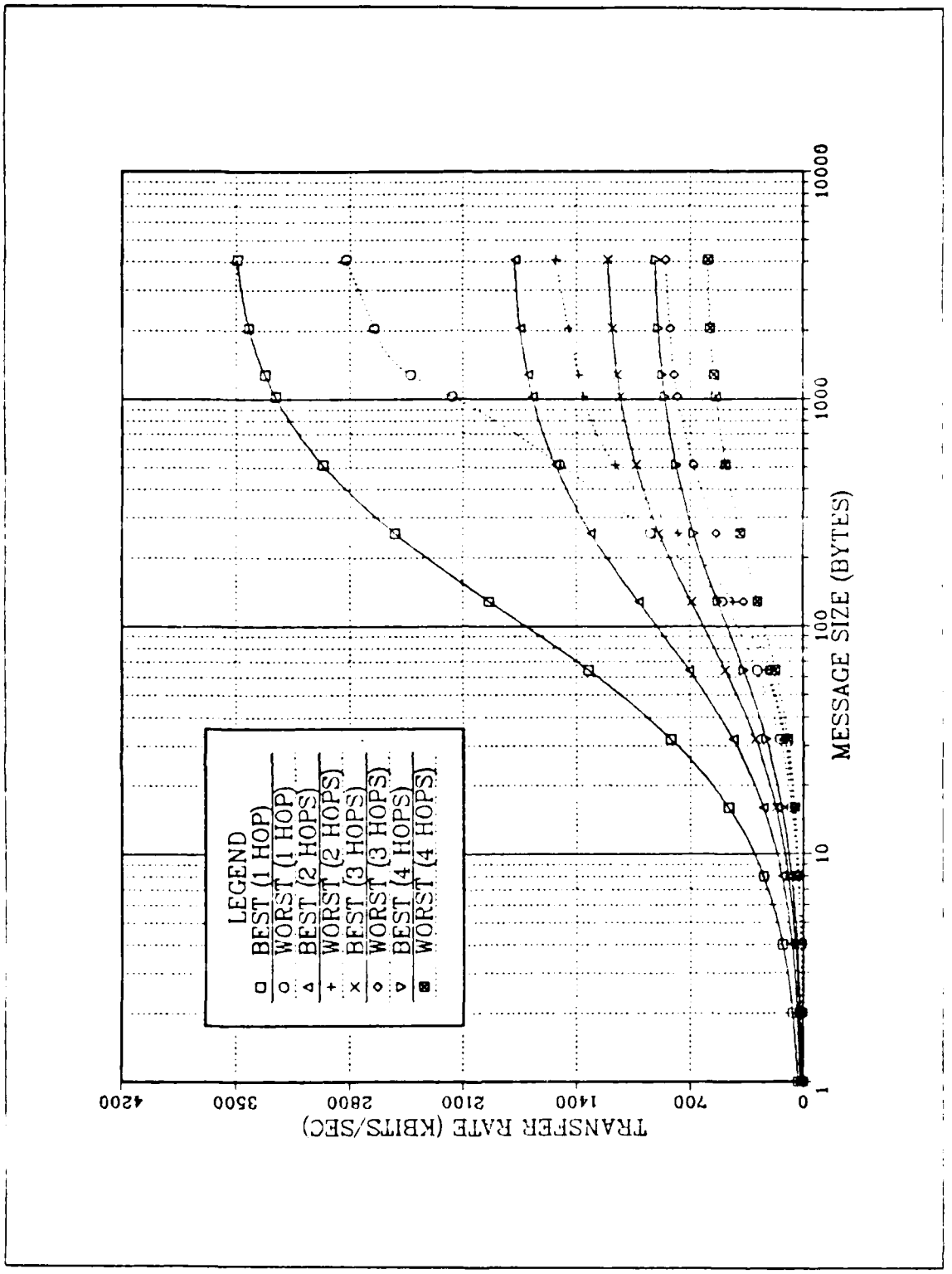


Figure 5.4 Transfer Rates with Multiple Retransmissions.

small messages. However, as can be seen in Table 7 and Figure 5.5, the effect of the header size was found to be minimum for all message sizes. In the worst case, we cannot even notice any difference at all.

The new protocol we will be testing, will have a header which is 3 bytes[17] long, and the only difference from the previous one, is that the message size information will take up one byte, rather than two. Therefore, we will be limited to messages up to 255 bytes.

TABLE 7

TRANSFER RATES WITH THE NEW HEADER
BETWEEN ADJACENT TRANSPUTERS (KBITS/SEC)

BYTES	1OUT	1IN	2OUT	2IN	3OUT	3IN	4OUT	4IN	4INOUT
1	22	38	11	32	7	14	5	11	4
2	44	76	23	64	14	29	10	22	9
4	89	151	56	129	27	59	21	45	19
8	174	293	92	251	55	116	43	88	38
16	332	540	179	469	110	225	86	172	76
32	613	939	341	821	215	419	170	330	151
64	1051	1486	626	1329	406	748	324	590	297
128	1634	2092	1106	1913	732	1211	597	1051	527
255	2266	2648	1697	2456	1216	1763	1026	1518	957

E. A CONTROVERSIAL PROBLEM

In the first version of our evaluation program, we were using the remote transputers, which are T414 (15 Mhz), to send the synchronization flags to the root transputer, which is a T414 (12.5 MHz). We had chosen this way because we would be able to start timing only after the flag had passed through the operating system, resulting in a more accurate timing.

However, after a period of approximately 16 seconds the program was deadlocking in a random state, and bear in mind that within this time, we were able to perform up to 9 complete runs of the same program ! Another symptom was that before deadlocking, we could notice a very big increase in the transfer rates of the "INs".

[17]We have chosen the new header size to be 3 bytes long, because in doing so, we wouldn't have to make major changes in the protocol design.

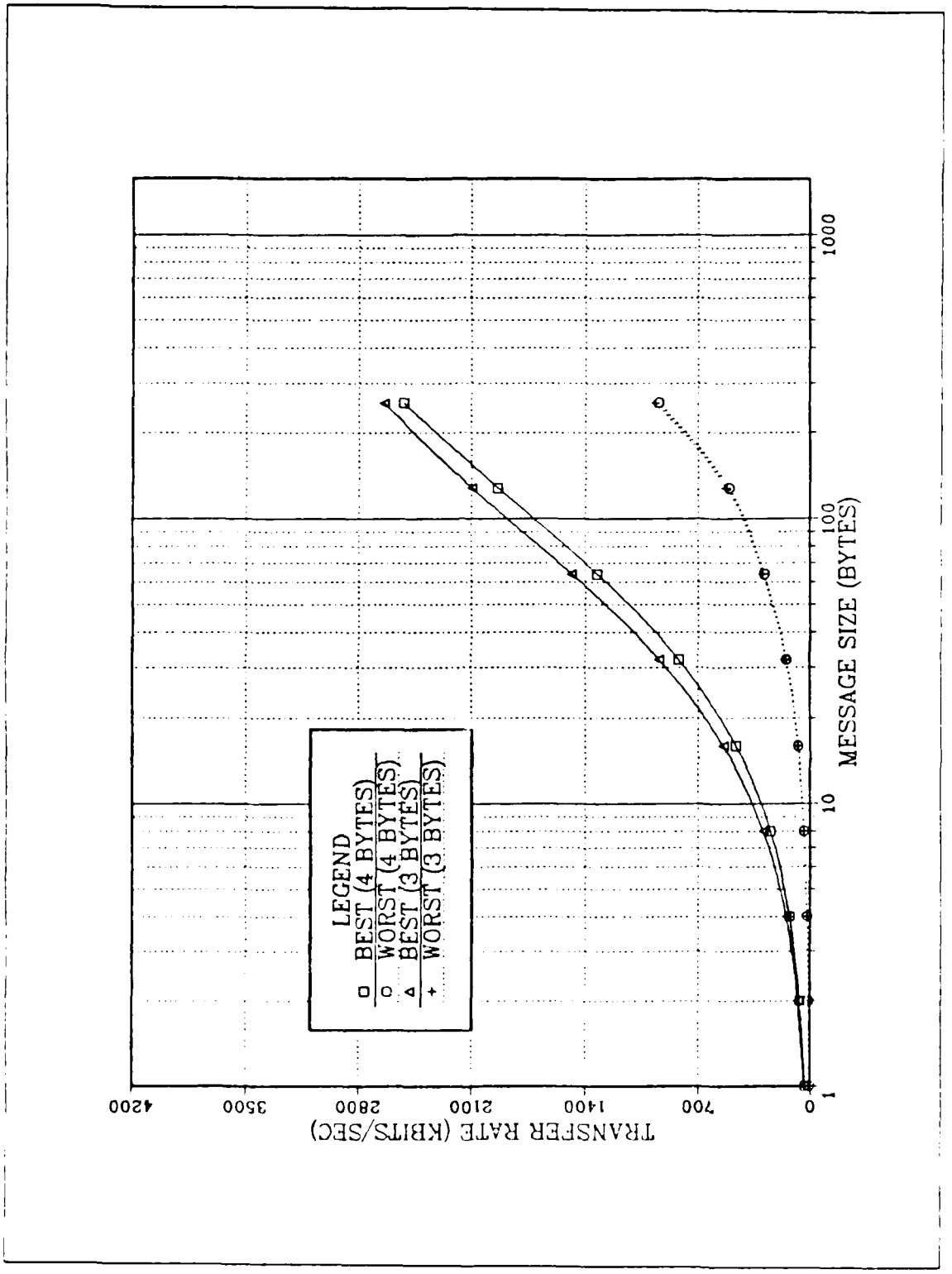


Figure 5.5 Effect of the Header Size in the Transfer Rate.

The reasons for such a behavior are still unknown, but we found it would be useful to pass this information, since it may happen again, in some other experiment.

VI. USING THE OPERATING SYSTEM

A. INTRODUCTION

This Chapter will introduce the operating system under the user's point of view. Our system is intended to be used in any transputer network, but when dealing with very small networks, up to 5 transputers, where we can always find a direct communication path between them, it may be a better choice not using the whole operating system, but just its library routines, mainly when dealing with small messages and when efficiency is a critical issue. However, in all other cases we strongly suggest its use, so that the debugging capability of the distributed system will be extremely enhanced, as well as the ease of programming.

B. THE REQUIRED PROGRAM STRUCTURE

The very first steps are basically the same for programs with or without the operating system, and they are:

1. Divide the entire program into modules, which can be run in separate processors. Ultimately, these modules will become SCs to be placed in our configuration.
2. Specify the interfaces between modules (or SCs) and make a sketch of the desired configuration.

Now it is time to change a little bit the structure of the TDS program, as presented in Chapter II.

The new structure is the one presented in Figure 6.1, where some comments are required for a better understanding.

As one can see, what we need to do is to create a filed fold containing either the root or the remote operating system source code,[18] in each of the SCs, and run the user process in parallel with the operating system. As discussed in the previous Chapter, the PRI PAR construct should be more efficient, but we suggest to develop the entire program with no priority assignments, leaving them for the very end.

Another peculiarity is the new parameters list which must be passed in the configuration part. The channels parameters have been extensively discussed in Chapter II, when we were talking about configuration, so that at this time, we will skip

[18]They are contained in the files ROOT_OS.TDS and REMOTE_OS.TDS respectively.

```
SC PROC transputer.root (CHAN A,B,C,D,E,F,G,H,
                         VALUE this.transputer,
                               t0,t1,t2,t3,t4,t5,t6,
                               t7,t8,t9,t10,t11,t12,
                               t13,t14,t15,t16,t17 )
 ...F  ROOT_OS.TDS
 ...   PROC user.root
   PAR
     operating.system
     user.root:

SC PROC transputer.1 (CHAN A,B,C,D,E,F,G,H,
                      VALUE this.transputer,
                            t0,t1,t2,t3,t4,t5,t6,
                            t7,t8,t9,t10,t11,t12,
                            t13,t14,t15,t16,t17 )
 ...F  REMOTE_OS.TDS
 ...   PROC user.1
   PAR
     operating.system
     user.1:

SC PROC transputer.2 (CHAN A,B,C,D,E,F,G,H,
                      VALUE this.transputer,
                            t0,t1,t2,t3,t4,t5,t6,
                            t7,t8,t9,t10,t11,t12,
                            t13,t14,t15,t16,t17 )
 ...F  REMOTE_OS.TDS
 ...   PROC user.2
   PAR
     operating.system
     user.2:

SC PROC transputer.n (CHAN A,B,C,D,E,F,G,H,
                      VALUE this.transputer,
                            t0,t1,t2,t3,t4,t5,t6,
                            t7,t8,t9,t10,t11,t12,
                            t13,t14,t15,t16,t17 )
 ...F  REMOTE_OS.TDS
 ...   PROC user.n
   PAR
     operating.system
     user.n:

... configuration
```

Figure 6.1 The Program Structure when using the Operating System.

them. The next parameter to be passed is the transputer id number of that specific transputer. This id number will be used as an index in the routing table, to find the output link to some message. It must be in the range of 0 to 17, since our actual implementation of the routing table has only 18 entries. Another suggestion is to use the same number used in the configuration, although it is not mandatory.

Finally, the last 18 parameters are the routing table itself, which was already extensively discussed in a previous Section in Chapter IV. The reason we are passing value by value, is just because OCCAM1 does not support TABLE data type as a parameter in the configuration. It is believed that this problem has been overcome in OCCAM2.

The user "cannot" change the names of the parameters this.transputer and t0 up to t17, since they will be passed as predefined global constants to the Operating System. These names must be the same for all SCs in the entire program.

C. PROGRAMMING WITH THE OPERATING SYSTEM

The art of programming a distributed system is usually done by lead programmers, with a great knowledge of the architecture they are working with. However, in our case, with the aid of the operating system, this task will be so much simplified, that it will be possible to an applications programmer to carry out this job.

However, as in any operating system, there are some peculiarities, which must be known by the user, before he starts to use it, and in our case they are the following:

- Whenever we want to communicate with external transputers we must use the "send" or "receive" routines. We also maintain in our Library a series of I/O routines and some utilities, which make use of the send routine and must be used only if we want to have some kind of output to the screen. Therefore, when using these library routines, we must always have the root transputer as our final destination.
- The user must have a complete knowledge, of the available library routines for the root, and for a remote transputer as well. We have tried to keep the same names for the procedures in both libraries, just adding a "rem" in front of the original name, if it was to be used in a remote transputer. The only exception was the "write.string", which was taken out from the remote library, since the "send" routine performs the same task, with even some more enhancements.
- If for some reason, none of the library routines fits our needs, we can still remotely access the screen, by using the "send" with the special channel "40" or "scrn".
- In the actual implementation, the valid channel ids are multiples of 10, starting from 50 up to 240.
- It is very important to keep in mind that for every "send" operation, must exist a matched "receive", with the same channel id number, and in the same destination transputer of the "send".

As one can see, there is no difficulty in programming with the operating system. In very few words, what needs to be done, is nothing else than a standard uniprocessor

program, and wherever we need some sort of external communication, either to the screen, or to any other transputer, we just have to follow the previous rules. As an example, we will provide in Appendix F, a complete listing of the evaluation program, which was running under the Operating System. In this program we make very little use of the various library routines available for remote transputers, but the main idea is just to show the overall structure of a program running under the Operating System.

D. ADVANTAGES OF THE OPERATING SYSTEM

We will now provide a list, with some of the main benefits, originated from the use of this operating system:

1) We can have as many as needed "send" or "receive" calls inside a parallel construct. The various "send" could be even for the same transputer, in which case, the operating system would take care of multiplexing them, through the correct output link. The only requirement is that all the channel ids in each of the send inside a parallel construct must be different.

2) Now we have the capability of debugging remote transputers. For example, we can send a unique flag to the screen when entering or exiting every procedure running in some remote transputer, so that we could obtain a complete trace of our program, and even determine where the system was deadlocking, if that was the case.

3) Thanks to the remote dump routine we can now dump the entire memory of any transputer in the network.

4) With the remote I/O routines, we have got formatting capability, so that we could have, for example, transputer 1 using the upper left part of the screen, transputer #2 using the middle part, transputer #3 the lower right part, and so forth.

E. CUSTOMIZING THE OPERATING SYSTEM

This Section describes which set up must be performed, prior to the use of this operating system with some user program.

There is a fold called "Operating System Global Declarations", which is the only place, where the user should perform any sort of change in the O.S. file. However, there are some declarations and definitions in there, which just need to be modified for maintenance purposes, done by qualified people.

In most cases, the only definition we will need to change is the "max.block.size", which specifies the size, in bytes, of the lengthiest message to be accepted in that network.

VII. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

This thesis is a first effort in developing an Operating System for a distributed system of transputers. It was built from three basic modules, namely, Input Handler, Output Handler and Screen Handler, which were implemented under severe efficiency requirements.

In order to achieve this basic requirement, we had to avoid moving data inside our system, as much as we could. A close look at the code will confirm, that most of the processing time is spent in exchanging flags among the different modules inside the operating system. Even the flags were checked for efficiency, in other words, the type of flag we are using is proved to be the fastest one in that regard. As the reader can notice, most of the flags and I/O are implemented with the BYTE.SLICE construct, which has been proved to be the fastest construct available in the transputer [Ref. 9]. Only in the alternate construct we keep the standard OCCAM I/O channels, because the "ALT" does not accept the BYTE.SLICE as a valid guard channel.

Now, as we may recall from Chapter V, the performance figures of the operating system, demonstrates that it is quite efficient, mainly for messages bigger than 2048 bytes. At this size for example, we have losses ranging from 8% up to 25%, depending on the number of channels, which are transmitting and receiving in parallel.[19] The last value was obtained when the 4 channels were transmitting and receiving in parallel.

As far as debugging goes, the operating system was very successful, in the sense that it brought up a new perspective in this field, for a distributed system of transputers. Although we realize that it is still far from being ideal, we should agree that it provides the user with some capabilities, which were not easily achieved up to now.

The other main goal to be achieved by this thesis, is regarding the ease of programming a distributed system of transputers. Now, anyone would be able to very quickly, make a program to be run in a very large transputer network.

---

[19]This figures can be increased, mainly when dealing with smaller messages, if we decide to use the operating system in high priority, as depicted in Chapter V (Table 3).

Returning to the efficiency issue, which is one of the major concerns in real time, we strongly believe that if the present trend of increasing transmission speed continues, we will be reaching a point where no more shared memory will be needed, since the link speeds will be sufficient high, to allow the transmission of the global shared data to all users of it. Let us put some numbers in this previous assumption. Assuming that the new family of transputers, the T-800's, will really support a truly 30 MBits/sec for the link transmission speed, we will be able to achieve with our basic operating system, after 4 retransmissions, and also considering the worst case (4INOUT), rates of the order of 5 MBits/sec, which is a fairly high rate.

Two conclusions can arise from the preliminary results. First, a real time operating system for a network of transputers is feasible and highly recommended. Second, the shared memory architecture seems to be no longer the preferable architecture for sharing global data, since we are going to be able to achieve comparable results, without the disadvantages of having shared memory, like for example, the system bus constituting a single point of failure.

Therefore, the transputer appears to be an attractive architecture for implementing real time applications, where the reliability is a fundamental issue.

B. RECOMMENDED FOLLOW-ON WORK

As stated in Chapter I, this thesis is a first approach to a basic operating system for a network of transputers, and it is our hope that it serves as a firm foundation for future and more enhanced implementations.

As this thesis was being developed, many new ideas were brought up, and in this Section we will try to give some suggestions for future enhancements in the system:

- Conversion of all programs used in this thesis to OCCAM2. It is also important, to reevaluate this new version of the operating system.
- Implementation of some filer routines which would allow one to open, close, read and write to VMS files. However, it is believed that OCCAM2 provides already this capability.
- Creation of one more reserved channel in the Operating System, which could handle inputs from the keyboard to remote transputers, with a minimum interference in the process running in the root transputer. The suggestion is to make up a simple protocol for entering data from the keyboard, for example, always entering with the destination transputer id# first, followed by a carriage return, and only then, we should enter with the actual input data. The next step should be to change the procedure "read", which is inside the terminal driver, and insert a check for the transputer id#. If after checking, it was found to be for a remote transputer, we should send the incoming message directly to the

remote transputer, through a new reserved channel, in such a way, that the remote transputer would be able to recognize that the message was carrying some keyboard input data.

- Implementation of an adaptive routing to replace the present one, which is static. This feature could be further extended, in order to generate a complete fault tolerant system. OCCAM2 provides some built-in procedures like "OutputorFail", InputorFail" and "Reinitialise", which might be very helpful in solving this problem [Ref. 11].
- Construction of a more powerful set of Library Routines for the root and remote transputers, e.g. concatenation,....
- Make the Operating System resident in the transputers, in other words, when we turn the power on, it should be automatically loaded into all transputers. To accomplish this step we would need to change the loader program, which resides in the EPROM of the B001 board.
- Construction of a more powerful debugger. However, it is not imperative, in our understanding, to make a debugger with multiprocessor capability, since we can always map our program to run into a single transputer. In order to implement a debugger, we would have to make it resident in the upper part of memory, and we would also need to have some kind of deassembler, in order to correctly place the breakpoints inside the code.
- Enhancement of the Terminal Driver by changing its I/O handling. Presently, it is implemented by standard OCCAM I/O channels, so, the idea is to use the BYTE.SLICE, which is a much faster construct. However, keep in mind in mind that the single character will have to be handled as a special case, since the BYTE.SLICE only supports byte arrays.

APPENDIX A

OPS GLOBAL DEFINITIONS (GLOBAL_DEF.OPS)

```
-- global_def.ops
-- Constant Definitions
DEF port     = 0 :     --- assigns the RS232 port to the terminal
DEF baud     = 11:     --- set baud.rate to 9600 bps
DEF nul      = 0 :     --- null ascii value
DEF bell     = 7 :     --- bell ascii value
DEF tab      = 9 :     --- tab ascii value (every 8 col)
DEF lf       = 10:     --- linefeed ascii value
DEF cr       = 13:     --- carriage return ascii value
DEF esc      = 27:     --- escape ascii value
DEF sp       = 32:     --- space ascii value
-- Channel Declarations
CHAN Parameters AT  0 :
CHAN Screen     AT  1 :
CHAN Keyboard   AT  2 :
CHAN Filein0    AT  3 :
CHAN Filein1    AT  4 :
CHAN Filein2    AT  5 :
CHAN Filein3    AT  6 :
CHAN Filein4    AT  7 :
CHAN Filein5    AT  8 :
CHAN Filein6    AT  9 :
CHAN Filein7    AT 10 :
CHAN Fileout0   AT 11 :
CHAN Fileout1   AT 12 :
CHAN Fileout2   AT 13 :
CHAN Fileout3   AT 14 :
CHAN Fileout4   AT 15 :
CHAN Fileout5   AT 16 :
CHAN Fileout6   AT 17 :
CHAN Fileout7   AT 18 :
-- Link Definitions
DEF link0out = 0 :
DEF link1out = 1 :
DEF link2out = 2 :
DEF link3out = 3 :
DEF link0in  = 4 :
DEF link1in  = 5 :
DEF link2in  = 6 :
DEF link3in  = 7 :
-- File Handler Control Values
DEF ClosedOK            =  -1 :
DEF CloseFile           =  -2 :
DEF EndBuffer           =  -3 :
DEF EndFile             =  -4 :
DEF EndName             =  -5 :
DEF EndParameterString  =  -6 :
DEF EndRecord           =  -7 :
DEF NextRecord          =  -9 :
DEF OpenedOK            = -10 :
DEF OpenForRead         = -11 :
DEF OpenForWrite        = -12 :
```

```
-- File Handler Error Values
DEF FileNameTooLong       = #80000000 :
DEF InputFileNotOpened    = #80000001 :
DEF OutputFileNotCreated  = #80000002 :
DEF InputRecordTooLong    = #80000004 :
DEF ReadFailed            = #80000008 :
DEF OutputRecordTooLong   = #80000010 :
DEF WriteFailed           = #80000020 :
DEF CloseFailed           = #80000040 :
```

APPENDIX B

TDS GLOBAL DEFINITIONS (GLOBAL_DEF.TDS)

```
-- global_def.tds
-- Constant Definitions
DEF port     = 0 :     --- assigns the RS232 port to the terminal
DEF baud     = 11:     --- set baud.rate to 9600 bps
DEF nul      = 0 :     --- null ascii value
DEF bell     = 7 :     --- bell ascii value
DEF tab      = 9 :     --- tab ascii value (every 8 col)
DEF lf       = 10:     --- linefeed ascii value
DEF cr       = 13:     --- carriage return ascii value
DEF esc      = 27:     --- escape ascii value
DEF sp       = 32:     --- space ascii value

-- Channel Declarations
CHAN Screen            :
CHAN Keyboard          :

-- Link Definitions
DEF link0out = 0 :
DEF link1out = 1 :
DEF link2out = 2 :
DEF link3out = 3 :
DEF link0in  = 4 :
DEF link1in  = 5 :
DEF link2in  = 6 :
DEF link3in  = 7 :
```

APPENDIX C

TDS LIBRARY ROUTINES WITHOUT OPERATING SYSTEM (LIBRARY.TDS)

```
--- ************************************************************
--- *   Title:  LIBRARY.TDS                    * Version: 1.0  *
--- *   Author: MAURICIO DE MENEZES CORDEIRO   * Mod: 0        *
--- *   Date:   19/FEB/1987                    *****************
--- *   Programming Language:  OCCAM 1                         *
--- *   Compiler:  IMS D-600 (VAX/VMS)                         *
--- *   Brief Description: This program contains some library  *
--- *   routines to be used in any TDS program, when not       *
--- *   using the Operating System. It must be placed in       *
--- *   parallel with the user process and with the global     *
--- *   definitions for TDS.                                   *
--- ************************************************************
--- *   Mod #:       Date:                                     *
--- *   Responsible:                                           *
--- *   Brief Description:                                     *
--- *                                                          *
--- ************************************************************
--- *   Mod #:       Date:                                     *
--- *   Responsible:                                           *
--- *   Brief Description:                                     *
--- *                                                          *
--- ************************************************************
-- io_routines
-- PROC dec.to.hex (VALUE integer, VAR string[])
--- ************************************************************
--- DESCRIPTION: It converts an integer number from its        *
--- decimal representation into the equivalent hexadecimal     *
--- one. It accepts any valid integer. It returns the          *
--- hexadecimal number stored in a string of 10 bytes long     *
--- where the leading zeros are preserved.                     *
--- It returns the following format: [size]#0000FFFF           *
--- USAGE: dec.to.hex(37182,hex.string)                        *
--- REMARK: The BYTE[0] of the string carries its length       *
--- which is always 9, therefore it could be deleted, but      *
--- we decided to keep it.                                     *
--- ************************************************************ *

PROC dec.to.hex (VALUE integer, VAR string []) =
  VAR first, order.of.digit, digit :
  VAR number :
  DEF hex.char = "0123456789ABCDEF" :
  SEQ
    first := TRUE
    string [BYTE 0] := 9
    string [BYTE 1] := '#'
    number := integer
    order.of.digit := 9
    WHILE (number > 0) OR (first=TRUE)
      SEQ
        digit := number /\ #F
        digit := hex.char [BYTE digit + 1]
        string [BYTE order.of.digit] := digit
        number := number >> 4
        order.of.digit := order.of.digit - 1
        first := FALSE
    SEQ i = [2 FOR (order.of.digit - 1)]
      string [BYTE i] := '0':


-- PROC dec.to.ascii (VALUE integer, VAR string [])
```

```
--- ************************************************************
--- DESCRIPTION: It converts an integer number from its          *
--- decimal representation into the equivalent ASCII one. It     *
--- accepts any valid integer number. It returns the ASCII       *
--- number stored in a string of 12 bytes long, where the        *
--- number is right justified and it has the following           *
--- format:  - 3542  ---> '-'' '' '' '' '' '' ''3''5''4''2'     *
---          1922937 ---> ' '' '' '' ''1''9''2''2''9''3''7'     *
--- USAGE: dec.to.ascii(-9873,ascii.string)                      *
--- REMARK: The BYTE[0] of the string carries its length         *
--- which is always 11, therefore it could be eliminated, but    *
--- we decided to keep it.                                       *
--- ************************************************************

PROC dec.to.ascii (VALUE integer, VAR string []) =
  VAR number :
  VAR order.of.digit :
  DEF min.int = - 2147483648 :

  SEQ
    number := integer
    order.of.digit := 11
    string [BYTE 0] := 11
    IF
      number = min.int
        -- taking care of the limit case
        SEQ
          string [BYTE 1]  := '-'
          string [BYTE 2]  := '2'
          string [BYTE 3]  := '1'
          string [BYTE 4]  := '4'
          string [BYTE 5]  := '7'
          string [BYTE 6]  := '4'
          string [BYTE 7]  := '8'
          string [BYTE 8]  := '3'
          string [BYTE 9]  := '6'
          string [BYTE 10] := '4'
          string [BYTE 11] := '8'

      TRUE
        SEQ
          IF
            number = 0
              SEQ
                string [BYTE 1] := ' '
                string [BYTE 11]:= '0'
                order.of.digit := 10
            number < 0
              SEQ
                number := - number
                string [BYTE 1] := '-'
            TRUE                                  --- number > 0
              string [BYTE 1] := ' '
          -- building up the actual number
          WHILE number > 0
            SEQ
              string [BYTE order.of.digit] := (number \ 10) + '0'
              number := (number / 10)
              order.of.digit := order.of.digit - 1

          SEQ i = [2 FOR (order.of.digit - 1)]
            string [BYTE i] := ' ':


-- PROC hex.to.dec (VALUE string[], VAR integer, OK)
--- ************************************************************
--- DESCRIPTION: It accepts a hexadecimal representation of      *
---  a number and converts it into an integer number. It         *
--- expects the byte[0] of the string to carry the size          *
--- information of that "hex number".                            *
```

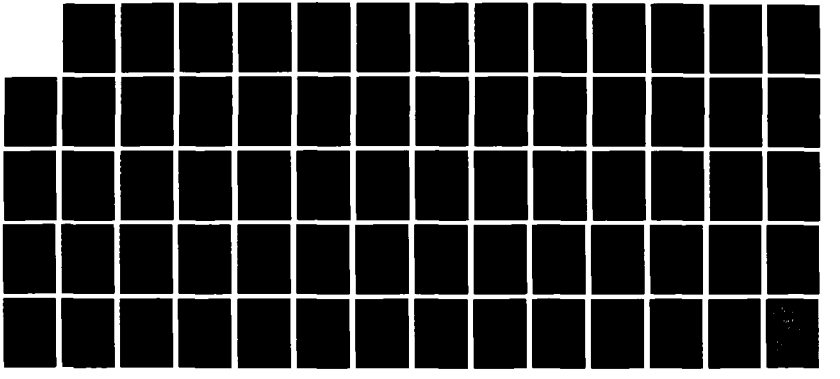
```
--- USAGE: hex.to.dec ("#00003785",number,valid)                    *
---        hex.to.dec ("#1452",number,valid)                        *
---        hex.to.dec ("#19574",number,valid)                       *
---        ascii.to.dec (hex.string,number,valid)                   *
--- REMARK: Returns a boolean value FALSE in OK if the             *
--- string is not in the correct format.                           *
--- ***************************************************************

PROC hex.to.dec (VALUE string [], VAR integer, OK) =
  SEQ
    integer := 0
    IF
      -- empty string
      string [BYTE 0] = 0
        OK := FALSE

      -- hex number
      string [BYTE 0] <> 0
        IF
          -- starts with '#'
          string [BYTE 1] = '#'
            VAR Count :
            SEQ
              OK := TRUE
              Count := 2
              WHILE (Count <= string [BYTE 0]) AND OK
                VAR Digit :
                SEQ
                  DEF hexChars = "0123456789ABCDEF" :
                  IF
                    IF Index = [1 FOR hexChars [BYTE 0]]
                      hexChars [BYTE Index] = string [BYTE Count]
                        Digit := Index - 1
                    TRUE
                      OK := FALSE
                  integer := (integer << 4) + Digit
                  Count := Count + 1

          -- otherwise
          string [BYTE 1] <> '#'
            OK := FALSE
    SKIP :


-- PROC ascii.to.dec (VALUE string[], VAR integer, OK)
--- ***************************************************************
--- DESCRIPTION: It accepts an ascii decimal                       *
--- representation of a number and converts it into an             *
--- integer number. It expects the byte[0] of the string           *
--- to carry the size information of that "ascii number".          *
--- USAGE: ascii.to.dec ("-3785",number,valid)                     *
---        ascii.to.dec ("+1452",number,valid)                     *
---        ascii.to.dec ("19574",number,valid)                     *
---        ascii.to.dec (string,number,valid)                      *
--- REMARK: Returns a boolean value FALSE in OK if the             *
--- string is not in the correct format.                           *
--- ***************************************************************

PROC ascii.to.dec (VALUE string [], VAR integer, OK) =
  SEQ
    integer := 0
    IF
      -- empty string
      string [BYTE 0] = 0
        OK := FALSE

      -- number
      string [BYTE 0] <> 0
        VAR Sign :
        VAR Start :
```

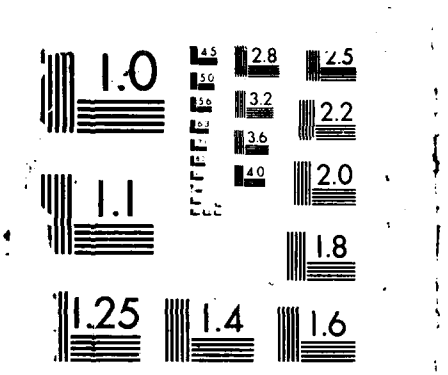
```
        VAR Length :
        SEQ
          OK := TRUE
          IF
            -- negative
            string [BYTE 1] = '-'
              SEQ
                Sign := - 1
                Start := 2
                Length := string [BYTE 0] - 1

            -- positive
            string [BYTE 1] <> '-'
              SEQ
                Sign := 1
                Start := 1
                Length := string [BYTE 0]

          -- convert to integer
          SEQ Index = [Start FOR Length]
            VAR Digit :
            SEQ
              Digit := string [BYTE Index]
              IF
                ('0' <= Digit) AND (Digit <= '9')
                  integer := (integer * 10) + (Digit - '0')
                TRUE
                  OK := FALSE

          integer := integer * Sign
    SKIP :


-- PROC write.string (VALUE string[])
--- ************************************************************
--- DESCRIPTION: Writes a given string to the screen, in a     *
--- byte by byte fashion. It requires that the string which    *
--- is a byte array, provides the size of the string in its    *
--- byte[0], otherwise we will get unpredictable results. We   *
--- are limited to strings up to 255 characters. For bigger    *
--- byte arrays or for partial printing use "send.string".     *
--- USAGE: write.string ("Hello")                              *
--- REMARK: It does not provide an automatic cr,lf.            *
--- ************************************************************

PROC write.string (VALUE string[]) =
  SEQ
    SEQ i = [1 FOR string[BYTE 0]]
      Screen ! string[BYTE i]
    SKIP:


-- PROC write.string.fast (VALUE string[])
--- ************************************************************
--- DESCRIPTION: This procedure works just in TDS and speeds   *
--- up things since the whole block is scheduled by CPU just   *
--- once, unlikely in the PROC write.string where each byte    *
--- is individually scheduled. However the terminal driver     *
--- routine MUST BE changed prior to the use of this routine.  *
--- USAGE: write.string.fast (string)                          *
--- ************************************************************

PROC write.string.fast (VALUE string[])=
  SEQ
    BYTE.SLICE.OUTPUT (Screen,string,1,string[BYTE 0]):


-- PROC write.number (VALUE integer)
--- ************************************************************
--- DESCRIPTION: This PROC outputs a signed integer value to *
```

```
--- the screen. It left justifies the number, so that if you *
--- need it right justified, use the dec.to.ascii and then  *
--- the write.string routines.                              *
--- It uses the following format:                           *
---                        0    --->  0                     *
---                  -234193    --->  -234193               *
---                     1496    --->  149                   *
--- USAGE: write.number (integer)                           *
---        write.number(135)                                *
--- ********************************************************

PROC write.number(VALUE integer) =
  DEF min.int = - 2147483648 :
  DEF max.digits = 11 :
  VAR number :
  VAR order.of.digit :
  VAR digit [BYTE 12] :

  SEQ
    number := integer
    order.of.digit := 11
    IF
      number = 0
        Screen ! '0'
      number = min.int
        Screen ! '-';'2';'1';'4';'7';'4';'8';'3';'6';'4';'8'
      TRUE
        SEQ
          IF
            number < 0
              SEQ
                number := - number
                Screen ! '-'
            TRUE                                --- number > 0
              SKIP
          WHILE number > 0
            SEQ
              digit [BYTE order.of.digit] := (number \ 10) + '0'
              number := (number / 10)
              order.of.digit := order.of.digit - 1
          SEQ i = [(order.of.digit + 1) FOR (max.digits-order.of.digit)]
            Screen ! digit [BYTE i]
          SKIP:


-- PROC read.string (VAR string[])
--- ********************************************************
--- DESCRIPTION: Reads any input sequence of characters typed*
--- from the keyboard and stores them in a string, while    *
--- echoing them to the screen. The PROC is exited when a   *
--- "cr" is typed.                                          *
--- USAGE: read.string (into.string)                        *
--- REMARK1: The byte[0] carries the size information of the *
--- string.                                                 *
--- REMARK2: Although it accepts strings of any length, the *
--- size contained in byte[0] will be reliable only for     *
--- strings up to 255 bytes.                                *
--- REMARK3: To enter with strings bigger than 80 bytes use *
--- the lf key in the keyboard.                             *
--- REMARK4: The set up of your cr key in your keyboard will*
--- determine where the cursor will be at the end of the    *
--- routine.                                                *
--- ********************************************************

PROC read.string (VAR string[]) =
  VAR n, char :
  SEQ
    char := 'z'
    n := 0
    WHILE char <> cr
```

AD-A186 593 DESIGN IMPLEMENTATION AND EVALUATION OF AN OPERATING SYSTEM FOR A NETWORK OF TRANSPUTERS(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA M D CORDEIRO 2/3

UNCLASSIFIED F/G 25/2 NL





```
      SEQ
        Keyboard ? char
        Screen ! char
        n := n + 1
        string[BYTE n] := char
    Screen ! lf
    string[BYTE 0] := n :


-- PROC read.number (VAR number)
--- ************************************************************
--- DESCRIPTION This procedure reads a number as entered from*
--- the keyboard. It accepts the following entries format:   *
---                          4536122 <cr>                    *
---                            +3782 <cr>                    *
---                          -573485 <cr>                    *
--- USAGE: read.number (into.integer)                        *
--- REMARK1: Only valid inputs will be echoed to the screen, *
--- so if you enter with -34r5&6, the following will         *
--- appear in the screen: -3456 meaning that the number      *
--- -3456 was accepted.                                      *
--- REMARK2: This procedure does not check to see if the     *
--- number is bigger than MAXINT or smaller than MININT. If  *
--- that happens the result will be incorrect.               *
--- REMARK3: An automatic cr,lf is provided when exiting.    *
--- ************************************************************

PROC read.number(VAR number) =
  VAR ch :
  VAR negative :
  SEQ
    ch := 'z'
    number := 0
    negative := FALSE
    WHILE (ch <> '-') AND (ch <> '+') AND ((ch < '0') OR (ch > '9'))
      Keyboard ? ch
    IF
      ch = '-'
        SEQ
          negative := TRUE
          Screen ! ch
      ch = '+'
        Screen ! ch
      TRUE
        SKIP
    WHILE ch <> cr
      SEQ
        WHILE (ch <> cr) AND ((ch < '0') OR (ch > '9'))
          Keyboard ? ch
        number := ( number * 10 ) + ( ch - '0' )
        Screen ! ch
        Keyboard ? ch
    SKIP
    Screen ! lf
    IF
      negative
        number := - number
      TRUE
        SKIP:


-- PROC clear.screen
--- ************************************************************
--- DESCRIPTION: It clears the screen and homes the cursor.  *
--- USAGE: clear.screen                                      *
--- ************************************************************

PROC clear.screen =
  SEQ
    Screen ! esc; '['; '2'; 'J'          --- clear screen sequence
```

```
    Screen ! esc; '['; 'H':                    --- home cursor

-- PROC pos.cursor (VALUE line, column)
--- ************************************************************
--- DESCRIPTION: Positions the cursor in a specified line and*
--- column. We have used the ANSI escape sequence             *
--- ESC [Line;Column H.                                       *
--- USAGE: pos.cursor (8,30)                                  *
--- REMARK1: Valid values for line   are 0 up to 24           *
---          Valid values for column are 0 up to 80           *
--- REMARK2: Values out of the above range will cause         *
---          unpredictable results.                           *
--- ************************************************************

PROC pos.cursor (VALUE line, column) =
  VAR x [BYTE 2], y [BYTE 2]:
  SEQ
    IF
      (line < 10) AND (line >= 0)
        SEQ
          y [BYTE 0] := '0'
          y [BYTE 1] := line + #30
      (line >= 10) AND (line <= 24)
        SEQ
          y [BYTE 0] := (line/10) + #30
          y [BYTE 1] := (line\10) + #30
      TRUE
        SKIP
    IF
      (column < 10) AND (column >= 0)
        SEQ
          x [BYTE 0] := '0'
          x [BYTE 1] := column + #30
      (column >= 10) AND (column <= 80)
        SEQ
          x [BYTE 0] := (column/10) + #30
          x [BYTE 1] := (column\10) + #30
      TRUE
        SKIP
    Screen ! esc; '[' ; y [BYTE 0]  ; y [BYTE 1]  ; ';' ;
                        x [BYTE 0]  ; x [BYTE 1]  ; 'H':


-- PROC new.line (VALUE number)
--- ************************************************************
--- DESCRIPTION: It will skip as many lines as specified in  *
--- its parameters list.                                      *
--- USAGE: new.line(4)                                        *
--- REMARK: Negative numbers will not give any new lines.     *
--- ************************************************************

PROC new.line (VALUE number) =
  SEQ
    SEQ i = [0 FOR number]
      Screen ! cr;lf
    SKIP:


-- PROC space (VALUE number)
--- ************************************************************
--- DESCRIPTION This procedure provides spaces for formatting*
--- a single line.                                            *
--- USAGE: space(8)                                           *
--- REMARK: This routine does not provide an automatic lf     *
--- after reaching the end of the line.                       *
--- ************************************************************

PROC space (VALUE number) =
  SEQ
```

```
    SEQ i = [0 FOR number]
      Screen ! sp
    SKIP:


-- PROC tab (VALUE number)
--- ************************************************************
--- DESCRIPTION This procedure provides tabs for formatting a*
--- single line. Each tab is equivalent to 8 spaces if the   *
--- terminal is using the default set up.                     *
--- USAGE: tab(6)                                             *
--- REMARK: This routine does not provide an automatic lf     *
--- after reaching the end of the line.                       *
--- ************************************************************

PROC tab (VALUE number) =
  SEQ
    SEQ i = [0 FOR number]
      Screen ! tab
    SKIP:


-- PROC send.string (CHAN output, VALUE string[],start,string.length)
--- ************************************************************
--- DESCRIPTION: This routine sends a string through a        *
--- generic channel output. It also allows to specify a start *
--- byte, as well as the length of the string to send.        *
--- USAGE: send.string (out.channel,"hello",3,3)              *
--- REMARK1: The above example will actually send the         *
--- characters 1,1 and o.                                     *
--- REMARK2: It can be used with the channel Screen as well.  *
--- ************************************************************

PROC send.string (CHAN output, VALUE string[],start,string.length) =
  SEQ
    SEQ index = [start FOR string.length]
      output ! string [BYTE index]
    SKIP:


-- PROC receive.string (CHAN input, VAR string[],
VALUE start,string.length)
--- ************************************************************
--- DESCRIPTION: This routine receives a string through a     *
--- generic channel input. It also allows to specify the      *
--- starting byte, as well as the number of bytes to receive  *
--- from the incoming string.                                 *
--- USAGE: receive.string (out.channel,string.in,3,2)         *
--- REMARK1: The above example will actually receive 2 bytes  *
--- from the incoming string, starting at byte 2.             *
--- ************************************************************

PROC receive.string (CHAN input, VAR string[],
VALUE start,string.length) =
  SEQ
    SEQ index = [start FOR string.length]
      input ? string [BYTE index]
    SKIP:


-- PROC send(VALUE channel.id,dest.transp,message[],start.byte,size)
--- ************************************************************
--- DESCRIPTION: It is an operating system routine, and it    *
--- is used to communicate between processors. It builds the  *
--- header of the message to be sent. It has as parameters    *
--- the channel id of the channel which is going to carry on  *
--- the communications, the id of the destination transputer  *
--- for that message, the start byte and the size of the      *
--- message to be transmitted. For every send must exist a    *
--- receive for that same channel id in the destination       *
```

```
--- transputer.                                                    *
--- USAGE: send (70,4,message,1,0)                                 *
--- REMARK: The user must be familiarized with the Operating       *
--- System Structure before using this routine.                    *
--- ***************************************************************

PROC send (VALUE channel.id,dest.transp,message[],start.byte,size)=
  VAR out,message.size,header [BYTE 5]:
  SEQ
    IF
      size <= 0  --- send from the start.byte all way to the end.
                 --- this method is valid for messages up to 255 bytes.
                 --- even for size < 0 it behaves like it was a 0.
        message.size := (message[BYTE 0] - start.byte) + 1
      TRUE
        message.size := size

    header [BYTE 1] := message.size/256 --- block.size (# of 256 bytes)
    header [BYTE 2] := message.size\256 ---              ( + remainder )
    header [BYTE 3] := channel.id   --- any tenth from 40 up to 240
    header [BYTE 4] := dest.transp  --- destination transputer
    out := route.table [dest.transp]
    BYTE.SLICE.OUTPUT (chan[channel.id + out],header,3,1) --- ready flag
    IF
      out = 4
        SEQ
          BYTE.SLICE.OUTPUT (link4,header,1,header.size)
          BYTE.SLICE.OUTPUT (link4,message,start.byte,message.size)
      out = 5
        SEQ
          BYTE.SLICE.OUTPUT (link5,header,1,header.size)
          BYTE.SLICE.OUTPUT (link5,message,start.byte,message.size)
      out = 6
        SEQ
          BYTE.SLICE.OUTPUT (link6,header,1,header.size)
          BYTE.SLICE.OUTPUT (link6,message,start.byte,message.size)
      out = 7
        SEQ
          BYTE.SLICE.OUTPUT (link7,header,1,header.size)
          BYTE.SLICE.OUTPUT (link7,message,start.byte,message.size)
    BYTE.SLICE.OUTPUT (chan[channel.id + out],header,3,1): --- done flag


-- PROC receive (VALUE channel.id,VAR message[], message.length[])
--- ***************************************************************
--- DESCRIPTION: It is an operating system routine, and it          *
--- is used to communicate between processors. It receives         *
--- the incoming message, and provides as an output parameter      *
--- the size of the message just received. The parameter           *
--- channel id must have an exact match with the send              *
--- operation which originated that message.                       *
--- USAGE: receive (70,message.in,size)                            *
--- REMARK1: The user must be familiarized with the Operating      *
--- System Structure before using this routine.                    *
--- REMARK2: Notice that the message.length output parameter,      *
--- must be a unity array of integers, while the message           *
--- itself must be declared as an array of bytes.                  *
--- ***************************************************************

PROC receive (VALUE channel.id,VAR message[],message.length[])=
  SEQ
    WORD.SLICE.INPUT (chan[channel.id],message.length,0,1)
    BYTE.SLICE.INPUT (chan[channel.id],message,1,message.length[0]):



-- utilities
-- PROC tick.to.time (VALUE start, stop, board.type)
--- ***************************************************************
--- DESCRIPTION: It expects the board type which can be :          *
```

```
---   board.type = 0 ----->  OPS (VAX VMS)                          *
---   board.type = 1 ----->  B001 (T414:12.5 MHz)                   *
---   board.type = 2 ----->  B002                                   *
---   board.type = 31----->  B003 (T414:15 MHz - high pri)          *
---   board.type = 32----->  B003 (T414:1 5MHz - low  pri)          *
---   board.type = 4 ----->  B004                                   *
--- and 2 signed integers representing some tick values            *
--- obtained by an assignment of the type TIME ? time.var         *
--- It then outputs the corrected elapsed time in hours, min,     *
--- sec and msec, already taking into account the fact that       *
--- the timer wraps around when it reaches MAXINT or MININT.      *
--- USAGE: tickk.to.time (time1,time2,31)                         *
--- REMARK: Although it takes care of the wrapping, it won't      *
--- keep track of the number of times you have completed one      *
--- full cycle of the timer. In order to solve this problem       *
--- you should record roughly the start time. For example, in     *
--- the VAX/VMS, the full cycle of the timer is 7.2 min, so       *
--- if you get the elapsed time of 5 min 7 sec 320 msec and       *
--- you have got a rough total time of 12 minutes, then the       *
--- real total time is 12 min 19 sec 320 msec.                    *
--- ***************************************************************

PROC tick.to.time (VALUE start, stop, board.type) =
  -- constant definitions
  DEF vax.sec     =10000000 :       --- hundreds of nsec/second
  DEF vax.mili    =   10000 :       --- hundreds of nsec/milisecond
  DEF b001.sec    =  625000 :       --- # of 1.6 usec/second
  DEF b001.mili   =     625 :       --- # of 1.6 usec/milisecond
  DEF b003h.sec   = 1000000 :       --- # of usec/second
  DEF b003h.mili  =    1000 :       --- # of usec/milisecond
  DEF b0031.sec   =   15625 :       --- # of 64 usec/second
  DEF b0031.mili  =      16 :       --- # of 64 usec/milisecond

  DEF max.number.of.ticks = 2147483648 : --- maximum integer (2**31)
  VAR elapsed.tick :
  VAR factor1, factor2 :
  VAR msec, tot.sec, sec, min, hr :

  SEQ
    IF
      board.type = 0                        --- VAX VMS
        SEQ
          factor1 := vax.sec
          factor2 := vax.mili

      board.type  = 1                       --- B001
        SEQ
          factor1 := b001.sec
          factor2 := b001.mili

      board.type  = 2                       --- B002
        SKIP                                --- not implemented

      board.type  = 31                      --- B003 in high priority
        SEQ
          factor1 := b003h.sec
          factor2 := b003h.mili

      board.type  = 32                      --- B003 in low priority
        SEQ
          factor1 := b0031.sec
          factor2 := b0031.mili

      board.type  = 4                       --- B004
        SKIP                                --- not implemented

    elapsed.tick := stop - start
    IF
      elapsed.tick < 0
        elapsed.tick := elapsed.tick + max.number.of.ticks
```

```
      TRUE
        SKIP

    tot.sec := elapsed.tick/factor1
    hr      := tot.sec/3600
    min     := (tot.sec\3600)/60
    sec     := tot.sec\60
    msec    := (elapsed.tick\factor1)/factor2

    -- output time to screen
    write.number (hr)
    write.string (" hr ")
    write.number (min)
    write.string (" min ")
    write.number(sec)
    write.string (" sec ")
    write.number(msec)
    write.string (" msec") :


-- PROC dump (VALUE begin.address, count)
--- ************************************************************
--- DESCRIPTION: This procedure dumps the memory starting at *
--- the given "begin.address". The value for the             *
--- "begin.address" can be either in hex or decimal.         *
--- The count value determines how many words in memory will *
--- be retrieved.                                            *
--- USAGE:           a) dump (#80003540,100)                 *
---                  b) dump (1024,48)                       *
---                  c) dump (-5113,1024)                    *
--- REMARK1: When specifying the count value remember that   *
--- the retrieval is done by words, not bytes!!!             *
--- REMARK2: If count is not a multiple of 4 it will use the *
--- closest upper multiple.                                  *
--- REMARK3: Negatives or zero values for count although     *
--- accepted, will give you no output.                       *
--- ************************************************************

PROC dump (VALUE begin.address, count) =
  VAR word.read:
  VAR hex.value [9], hex.addr[9]:
  VAR address, align, times:

  SEQ
    times := 0
    new.line(1)
    address := begin.address
    -- aligning a given address
    align := address\4
    IF
      align <> 0
        address := address - align
      TRUE
        SKIP


    WHILE times <= count
      SEQ
        write.string ("address ")
        dec.to.hex (address,hex.addr)
        write.string (hex.addr)
        write.string ("  -->  ")
        SEQ i = [0 FOR 4]
          SEQ
            GETWORD (word.read,address)
            dec.to.hex (word.read,hex.value)
            write.string (hex.value)
            space(2)
            times := times + 1
        SKIP
```

```
        address := address + 16
        new.line(1)
    SKIP:


-- PROC transfer.rate (VALUE start,stop,board.type,nr.of.bytes,VAR rate)
--- ***********************************************************
--- DESCRIPTION: It is basically the same routine as          *
--- tick.to.time, with the only difference that it returns a *
--- rate value in Kbits/sec instead of a time value.          *
--- USAGE: transfer.rate (time1,time2,31,4096,rate)           *
--- REMARK: If further information is needed, please refer to*
--- routine tick.to.time                                      *
--- ***********************************************************

PROC transfer.rate (VALUE start,stop,board.type,nr.of.bytes,VAR rate) =
  -- constant definitions
  DEF vax.sec    =10000000 :      --- hundreds of nsec/second
  DEF b001.sec   =  625000 :      --- # of 1.6 usec/second
  DEF b003h.sec  = 1000000 :      --- # of usec/second
  DEF b0031.sec  =   15625 :      --- # of 64 usec/second
  DEF max.number.of.ticks = 2147483648 : --- maximum integer (2**31)

  -- variable declarations
  VAR elapsed.tick :
  VAR factor :       --- to convert ticks to seconds

  SEQ
    elapsed.tick := stop - start
    IF
      elapsed.tick < 0
        elapsed.tick := elapsed.tick + max.number.of.ticks
      TRUE
        SKIP
    -- selection of correct factor iaw the board
    IF
      board.type = 0                        --- VAX VMS
        factor := vax.sec

      board.type  = 1                       --- B001
        factor := b001.sec

      board.type  = 2                       --- B002
        SKIP                                --- not implemented

      board.type  = 31                      --- B003 in high priority
        factor := b003h.sec

      board.type  = 32                      --- B003 in low priority
        factor := b0031.sec

      board.type  = 4                       --- B004
        SKIP                                --- not implemented

    -- rate calculation
    IF
      board.type = 32
        rate := ((nr.of.bytes*8)*factor)/(elapsed.tick*1000)
        --- operation is done this way to keep precision ok!
      TRUE
        rate := ((nr.of.bytes*8)*(factor/1000))/elapsed.tick
        --- operation is done this way in order not to exceed maxint
        --- on the numerator.
        --- multiply by 8 due to 8 bits per byte
        --- divide by 1000 to have the tranfer.rate in kbits/sec

    SKIP:

-- PROC capitalize (VAR ch[])
--- ***********************************************************
```

```
--- DESCRIPTION: It capitalizes the first character in any   *
--- string.                                                  *
--- USAGE: capitalize (string)                               *
--- *********************************************************

PROC capitalize (VAR ch[]) =

  DEF delta =('a' - 'A') :
                              ---  A ---> 65
                              ---  a ---> 97      ASCII values
                              ---  z ---> 122

  SEQ
    IF
      (ch [BYTE 1] <= 'z') AND (ch [BYTE 1] >= 'a')
        ch [BYTE 1] := ch [BYTE 1] - delta
      TRUE
        SKIP :
```

APPENDIX D

THE OPERATING SYSTEM FOR THE ROOT TRANSPUTER (ROOT_OS.TDS)

```
--- ********************************************************
--- *   Title:  ROOT_OS.TDS                    * Version: 1.0  *
--- *   Author: MAURICIO DE MENEZES CORDEIRO   * Mod: 0       *
--- *   Date:   21/MAR/1987                    ****************
--- *   Programming Language:  OCCAM 1                        *
--- *   Compiler:  IMS D-600 (VAX/VMS)                        *
--- *   Brief Description: This program contains the source   *
--- *   code for a communications operating system for the    *
--- *   root processor in a network of transputers. It must   *
--- *   be placed in parallel with the user process.          *
--- ********************************************************
--- *   Mod #:       Date:                                    *
--- *   Responsible:                                          *
--- *   Brief Description:                                    *
--- *                                                         *
--- ********************************************************
--- *   Mod #:       Date:                                    *
--- *   Responsible:                                          *
--- *   Brief Description:                                    *
--- *                                                         *
--- ********************************************************
-- Operating System global declarations
DEF max.block.size = 4100:
DEF nr.of.transputers = 17:
DEF header.size = 4:
DEF scrn = 40:                          --- channel screen
DEF max.io.channels = 25:               --- 0 up to 240, in tenths
DEF max.screen.channels = 5:
VAR route.table[18]:
VAR flag [BYTE 1]:                      --- for the library routines
CHAN chan [10 * max.io.channels]:       --- Actually it should be :
                                        ---(10*(max.io.channels-1))+8
CHAN screen [max.screen.channels]:


-- global_def.tds
--- ********************************************************
---      At this point we should imbed the filed fold       *
---    global_def.tds, which is described in Appendix B     *
--- ********************************************************


-- Operating System Channel Placements
CHAN link0 AT link0in :
CHAN link1 AT link1in :
CHAN link2 AT link2in :
CHAN link3 AT link3in :
CHAN link4 AT link0out:
CHAN link5 AT link1out:
CHAN link6 AT link2out:
CHAN link7 AT link3out:


-- root_lib.tds
-- io_routines
-- PROC dec.to.hex (VALUE integer, VAR string[])
--- ********************************************************
--- DESCRIPTION: It converts an integer number from its     *
--- decimal representation into the equivalent hexadecimal  *
--- one. It accepts any valid integer. It returns the       *
```

```
--- hexadecimal number stored in a string of 10 bytes long   *
--- where the leading zeros are preserved.                   *
--- It returns the following format: [size]#0000FFFF         *
--- USAGE: dec.to.hex(37182,hex.string)                      *
--- REMARK: The BYTE[0] of the string carries its length     *
--- which is always 9, therefore it could be deleted, but    *
--- we decided to keep it.                                   *
--- ******************************************************** *

PROC dec.to.hex (VALUE integer, VAR string []) =
  VAR first, order.of.digit, digit :
  VAR number :
  DEF hex.char = "0123456789ABCDEF" :
  SEQ
    first := TRUE
    string [BYTE 0] := 9
    string [BYTE 1] := '#'
    number := integer
    order.of.digit := 9
    WHILE (number > 0) OR (first=TRUE)
      SEQ
        digit := number /\ #F
        digit := hex.char [BYTE digit + 1]
        string [BYTE order.of.digit] := digit
        number := number >> 4
        order.of.digit := order.of.digit - 1
        first := FALSE
    SEQ i = [2 FOR (order.of.digit - 1)]
      string [BYTE i] := '0':


-- PROC dec.to.ascii (VALUE integer, VAR string [])
--- ********************************************************
--- DESCRIPTION: It converts an integer number from its      *
--- decimal representation into the equivalent ASCII one. It *
--- accepts any valid integer number. It returns the ASCII   *
--- number stored in a string of 12 bytes long, where the    *
--- number is right justified and it has the following       *
--- format:  - 3542  ---> '-'' '' '' '' '' '' ''3''5''4''2'  *
---          1922937 ---> ' '' '' '' ''1''9''2''2''9''3''7'  *
--- USAGE: dec.to.ascii(-9873,ascii.string)                  *
--- REMARK: The BYTE[0] of the string carries its length     *
--- which is always 11, therefore it could be eliminated, but*
--- we decided to keep it.                                   *
--- ********************************************************

PROC dec.to.ascii (VALUE integer, VAR string []) =
  VAR number :
  VAR order.of.digit :
  DEF min.int = - 2147483648 :

  SEQ
    number := integer
    order.of.digit := 11
    string [BYTE 0] := 11
    IF
      number = min.int
        -- taking care of the limit case
        SEQ
          string [BYTE 1]  := '-'
          string [BYTE 2]  := '2'
          string [BYTE 3]  := '1'
          string [BYTE 4]  := '4'
          string [BYTE 5]  := '7'
          string [BYTE 6]  := '4'
          string [BYTE 7]  := '8'
          string [BYTE 8]  := '3'
          string [BYTE 9]  := '6'
          string [BYTE 10] := '4'
          string [BYTE 11] := '8'
```

```
      TRUE
        SEQ
          IF
            number = 0
              SEQ
                string [BYTE 1] := ' '
                string [BYTE 11]:= '0'
                order.of.digit := 10
            number < 0
              SEQ
                number := - number
                string [BYTE 1] := '-'
            TRUE                              --- number > 0
              string [BYTE 1] := ' '
          -- building up the actual number
          WHILE number > 0
            SEQ
              string [BYTE order.of.digit] := (number \ 10) + '0'
              number := (number / 10)
              order.of.digit := order.of.digit - 1

          SEQ i = [2 FOR (order.of.digit - 1)]
            string [BYTE i] := ' ':


-- PROC hex.to.dec (VALUE string[], VAR integer, OK)
--- ************************************************************
--- DESCRIPTION: It accepts a hexadecimal representation of  *
---  a number and converts it into an integer number. It     *
--- expects the byte[0] of the string to carry the size       *
--- information of that "hex number".                         *
--- USAGE: hex.to.dec ("#00003785",number,valid)              *
---        hex.to.dec ("#1452",number,valid)                  *
---        hex.to.dec ("#19574",number,valid)                 *
---        ascii.to.dec (hex.string,number,valid)             *
--- REMARK: Returns a boolean value FALSE in OK if the        *
--- string is not in the correct format.                      *
--- ************************************************************

PROC hex.to.dec (VALUE string [], VAR integer, OK) =
  SEQ
    integer := 0
    IF
      -- empty string
      string [BYTE 0] = 0
        OK := FALSE

      -- hex number
      string [BYTE 0] <> 0
        IF
          -- starts with '#'
          string [BYTE 1] = '#'
            VAR Count :
            SEQ
              OK := TRUE
              Count := 2
              WHILE (Count <= string [BYTE 0]) AND OK
                VAR Digit :
                SEQ
                  DEF hexChars = "0123456789ABCDEF" :
                  IF
                    IF Index = [1 FOR hexChars [BYTE 0]]
                      hexChars [BYTE Index] = string [BYTE Count]
                        Digit := Index - 1
                    TRUE
                      OK := FALSE
                  integer := (integer << 4) + Digit
                  Count := Count + 1
```

```
        -- otherwise
        string [BYTE 1] <> '#'
          OK := FALSE
    SKIP :


-- PROC ascii.to.dec (VALUE string[], VAR integer, OK)
--- ***********************************************************
--- DESCRIPTION: It accepts an ascii decimal                    *
--- representation of a number and converts it into an          *
--- integer number. It expects the byte[0] of the string        *
--- to carry the size information of that "ascii number".       *
--- USAGE: ascii.to.dec ("-3785",number,valid)                  *
---        ascii.to.dec ("+1452",number,valid)                  *
---        ascii.to.dec ("19574",number,valid)                  *
---        ascii.to.dec (string,number,valid)                   *
--- REMARK: Returns a boolean value FALSE in OK if the          *
--- string is not in the correct format.                        *
--- ***********************************************************

PROC ascii.to.dec (VALUE string [], VAR integer, OK) =
  SEQ
    integer := 0
    IF
      -- empty string
      string [BYTE 0] = 0
        OK := FALSE

      -- number
      string [BYTE 0] <> 0
        VAR Sign :
        VAR Start :
        VAR Length :
        SEQ
          OK := TRUE
          IF
            -- negative
            string [BYTE 1] = '-'
              SEQ
                Sign := - 1
                Start := 2
                Length := string [BYTE 0] - 1

            -- positive
            string [BYTE 1] <> '-'
              SEQ
                Sign := 1
                Start := 1
                Length := string [BYTE 0]

          -- convert to integer
          SEQ Index = [Start FOR Length]
            VAR Digit :
            SEQ
              Digit := string [BYTE Index]
              IF
                ('0' <= Digit) AND (Digit <= '9')
                  integer := (integer * 10) + (Digit - '0')
                TRUE
                  OK := FALSE

          integer := integer * Sign

    SKIP :


-- PROC write.string (VALUE string[])
--- ***********************************************************
--- DESCRIPTION: Writes a given string to the screen, in a      *
--- byte by byte fashion. It requires that the string which     *
```

```
--- is a byte array, provides the size of the string in its  *
--- byte[0], otherwise we will get unpredictable results. We *
--- are limited to strings up to 255 characters. For bigger  *
--- byte arrays or for partial printing use "send.string".   *
--- USAGE: write.string ("Hello")                            *
--- REMARK: It does not provide an automatic cr,lf.          *
--- ********************************************************

PROC write.string (VALUE string[]) =
  SEQ
    BYTE.SLICE.OUTPUT (screen[4],flag,0,1)
    SEQ i = [1 FOR string[BYTE 0]]
      Screen ! string[BYTE i]
    BYTE.SLICE.OUTPUT (screen[4],flag,0,1) :


-- PROC write.string.fast (VALUE string[])
--- ********************************************************
--- DESCRIPTION: This procedure works just in TDS and speeds *
--- up things since the whole block is scheduled by CPU just *
--- once, unlikely in the PROC write.string where each byte  *
--- is individually scheduled. However the terminal driver   *
--- routine MUST BE changed prior to the use of this routine.*
--- USAGE: write.string.fast (string)                        *
--- ********************************************************

PROC write.string.fast (VALUE string[])=
  SEQ
    BYTE.SLICE.OUTPUT (screen[4],flag,0,1)
    BYTE.SLICE.OUTPUT (Screen,string,1,string[BYTE 0])
    BYTE.SLICE.OUTPUT (screen[4],flag,0,1) :


-- PROC write.number (VALUE integer)
--- ********************************************************
--- DESCRIPTION: This PROC outputs a signed integer value to *
--- the screen. It left justifies the number, so that if you *
--- need it right justified, use the dec.to.ascii and then   *
--- the write.string routines.                               *
--- It uses the following format:                            *
---                          0   --->  0                     *
---                  -234193   --->  -234193                 *
---                     1496   --->  149                     *
--- USAGE: write.number (integer)                            *
---        write.number(135)                                 *
--- ********************************************************

PROC write.number(VALUE integer) =
  DEF min.int = - 2147483648 :
  DEF max.digits = 11 :
  VAR number :
  VAR order.of.digit :
  VAR digit [BYTE 12] :

  SEQ
    number := integer
    order.of.digit := 11
    BYTE.SLICE.OUTPUT (screen[4],flag,0,1)
    IF
      number = 0
        Screen ! '0'
      number = min.int
        Screen ! '-';'2';'1';'4';'7';'4';'8';'3';'6';'4';'8'
      TRUE
        SEQ
          IF
            number < 0
              SEQ
                number := - number
                Screen ! '-'
```

```
      TRUE                                --- number > 0
        SKIP
    WHILE number > 0
      SEQ
        digit [BYTE order.of.digit] := (number \ 10) + '0'
        number := (number / 10)
        order.of.digit := order.of.digit - 1
    SEQ i = [(order.of.digit + 1) FOR (max.digits-order.of.digit)]
      Screen ! digit [BYTE i]
    BYTE.SLICE.OUTPUT (screen[4],flag,0,1) :


-- PROC read.string (VAR string[])
--- ************************************************************
--- DESCRIPTION: Reads any input sequence of characters typed*
--- from the keyboard and stores them in a string, while     *
--- echoing them to the screen. The PROC is exited when a    *
--- "cr" is typed.                                           *
--- USAGE: read.string (into.string)                         *
--- REMARK1: The byte[0] carries the size information of the *
--- string.                                                  *
--- REMARK2: Although it accepts strings of any length, the  *
--- size contained in byte[0] will be reliable only for      *
--- strings up to 255 bytes.                                 *
--- REMARK3: To enter with strings bigger than 80 bytes use  *
--- the lf key in the keyboard.                              *
--- REMARK4: The set up of your cr key in your keyboard will *
--- determine where the cursor will be at the end of the     *
--- routine.                                                 *
--- ************************************************************

PROC read.string (VAR string[]) =
  VAR n, char :
  SEQ
    char := 'z'
    n := 0
    BYTE.SLICE.OUTPUT (screen[4],flag,0,1)
    WHILE char <> cr
      SEQ
        Keyboard ? char
        Screen ! char
        n := n + 1
        string[BYTE n] := char
    Screen ! lf
    BYTE.SLICE.OUTPUT (screen[4],flag,0,1)
    string[BYTE 0] := n :


-- PROC read.number (VAR number)
--- ************************************************************
--- DESCRIPTION This procedure reads a number as entered from*
--- the keyboard. It accepts the following entries format:   *
---                        4536122 <cr>                      *
---                          +3782 <cr>                      *
---                        -573485 <cr>                      *
--- USAGE: read.number (into.integer)                        *
--- REMARK1: Only valid inputs will be echoed to the screen, *
--- so if you enter with -34r5&6, the following will         *
--- appear in the screen: -3456 meaning that the number      *
--- -3456 was accepted.                                      *
--- REMARK2: This procedure does not check to see if the     *
--- number is bigger than MAXINT or smaller than MININT. If  *
--- that happens the result will be incorrect.               *
--- REMARK3: An automatic cr,lf is provided when exiting.    *
--- ************************************************************

PROC read.number(VAR number) =
  VAR ch :
  VAR negative :
  SEQ
```

```
    ch := 'z'
    number := 0
    negative := FALSE
    BYTE.SLICE.OUTPUT (screen[4],flag,0,1)
    WHILE (ch <> '-') AND (ch <> '+') AND ((ch < '0') OR (ch > '9'))
      Keyboard ? ch
    IF
      ch = '-'
        SEQ
          negative := TRUE
          Screen ! ch
      ch = '+'
        Screen ! ch
      TRUE
        SKIP
    WHILE ch <> cr
      SEQ
        WHILE (ch <> cr) AND ((ch < '0') OR (ch > '9'))
          Keyboard ? ch
        number := ( number * 10 ) + ( ch - '0' )
        Screen ! ch
        Keyboard ? ch
    SKIP
    Screen ! lf
    IF
      negative
        number := - number
      TRUE
        SKIP
    BYTE.SLICE.OUTPUT (screen[4],flag,0,1)  :


-- PROC clear.screen
--- ************************************************************
--- DESCRIPTION: It clears the screen and homes the cursor.  *
--- USAGE: clear.screen                                      *
--- ************************************************************

PROC clear.screen =
  SEQ
    BYTE.SLICE.OUTPUT (screen[4],flag,0,1)
    Screen ! esc; '['; '2'; 'J'              --- clear screen sequence
    Screen ! esc; '['; 'H'                   --- home cursor
    BYTE.SLICE.OUTPUT (screen[4],flag,0,1) :


-- PROC pos.cursor (VALUE line, column)
--- ************************************************************
--- DESCRIPTION: Positions the cursor in a specified line and*
--- column. We have used the ANSI escape sequence            *
--- ESC [Line;Column H.                                      *
--- USAGE: pos.cursor (8,30)                                 *
--- REMARK1: Valid values for line   are 0 up to 24          *
---          Valid values for column are 0 up to 80          *
--- REMARK2: Values out of the above range will cause        *
---          unpredictable results.                          *
--- ************************************************************

PROC pos.cursor (VALUE line, column) =
  VAR x [BYTE 2], y [BYTE 2]:
  SEQ
    IF
      (line < 10) AND (line >= 0)
        SEQ
          y [BYTE 0] := '0'
          y [BYTE 1] := line + #30
      (line >= 10) AND (line <= 24)
        SEQ
          y [BYTE 0] := (line/10) + #30
          y [BYTE 1] := (line\10) + #30
```

```
      TRUE
        SKIP
    IF
      (column < 10) AND (column >= 0)
        SEQ
          x [BYTE 0] := '0'
          x [BYTE 1] := column + #30
      (column >= 10) AND (column <= 80)
        SEQ
          x [BYTE 0] := (column/10) + #30
          x [BYTE 1] := (column\10) + #30
      TRUE
        SKIP
    BYTE.SLICE.OUTPUT (screen[4],flag,0,1)
    Screen ! esc; '[' ; y [BYTE 0]  ; y [BYTE 1]  ; ';' ;
                        x [BYTE 0]  ; x [BYTE 1]  ; 'H'
    BYTE.SLICE.OUTPUT (screen[4],flag,0,1)  :


-- PROC new.line (VALUE number)
--- ************************************************************
--- DESCRIPTION: It will skip as many lines as specified in  *
--- its parameters list.                                     *
--- USAGE: new.line(4)                                       *
--- REMARK: Negative numbers will not give any new lines.    *
--- ************************************************************

PROC new.line (VALUE number) =
  SEQ
    BYTE.SLICE.OUTPUT (screen[4],flag,0,1)
    SEQ i = [0 FOR number]
      Screen ! cr;lf
    BYTE.SLICE.OUTPUT (screen[4],flag,0,1):


-- PROC space (VALUE number)
--- ************************************************************
--- DESCRIPTION This procedure provides spaces for formatting*
--- a single line.                                           *
--- USAGE: space(8)                                          *
--- REMARK: This routine does not provide an automatic lf    *
--- after reaching the end of the line.                      *
--- ************************************************************

PROC space (VALUE number) =
  SEQ
    BYTE.SLICE.OUTPUT (screen[4],flag,0,1)
    SEQ i = [0 FOR number]
      Screen ! sp
    BYTE.SLICE.OUTPUT (screen[4],flag,0,1):


-- PROC tab (VALUE number)
--- ************************************************************
--- DESCRIPTION This procedure provides tabs for formatting a*
--- single line. Each tab is equivalent to 8 spaces if the   *
--- terminal is using the default set up.                    *
--- USAGE: tab(6)                                            *
--- REMARK: This routine does not provide an automatic lf    *
--- after reaching the end of the line.                      *
--- ************************************************************

PROC tab (VALUE number) =
  SEQ
    BYTE.SLICE.OUTPUT (screen[4],flag,0,1)
    SEQ i = [0 FOR number]
      Screen ! tab
    BYTE.SLICE.OUTPUT (screen[4],flag,0,1):
```

```
-- PROC send.string (CHAN output, VALUE string[],start,string.length)
--- ************************************************************
--- DESCRIPTION: This routine sends a string through a          *
--- generic channel output. It also allows to specify a start   *
--- byte, as well as the length of the string to send.          *
--- USAGE: send.string (out.channel,"hello",3,3)                *
--- REMARK1: The above example will actually send the           *
--- characters l,l and o.                                       *
--- REMARK2: It can be used with the channel Screen as well.    *
--- ************************************************************

PROC send.string (CHAN output, VALUE string[],start,string.length) =
  SEQ
    SEQ index = [start FOR string.length]
      output ! string [BYTE index]
    SKIP:


-- PROC receive.string (CHAN input, VAR string[],
VALUE start,string.length)
--- ************************************************************
--- DESCRIPTION: This routine receives a string through a       *
--- generic channel input. It also allows to specify the        *
--- starting byte, as well as the number of bytes to receive    *
--- from the incoming string.                                   *
--- USAGE: receive.string (out.channel,string.in,3,2)           *
--- REMARK1: The above example will actually receive 2 bytes    *
--- from the incoming string, starting at byte 2.               *
--- ************************************************************

PROC receive.string (CHAN input, VAR string[],
VALUE start,string.length) =
  SEQ
    SEQ index = [start FOR string.length]
      input ? string [BYTE index]
    SKIP:


-- PROC send(VALUE channel.id,dest.transp,message[],start.byte,size)
--- ************************************************************
--- DESCRIPTION: It is an operating system routine, and it      *
--- is used to communicate between processors. It builds the    *
--- header of the message to be sent. It has as parameters      *
--- the channel id of the channel which is going to carry on    *
--- the communications, the id of the destination transputer    *
--- for that message, the start byte and the size of the        *
--- message to be transmitted. For every send must exist a      *
--- receive for that same channel id in the destination         *
--- transputer.                                                 *
--- USAGE: send (70,4,message,1,0)                              *
--- REMARK: The user must be familiarized with the Operating    *
--- System Structure before using this routine.                 *
--- ************************************************************

PROC send (VALUE channel.id,dest.transp,message[],start.byte,size)=
  VAR out,message.size,header [BYTE 5]:
  SEQ
    IF
      size <= 0  --- send from the start.byte all way to the end.
                 --- this method is valid for messages up to 255 bytes.
                 --- even for size < 0 it behaves like it was a 0.
        message.size := (message[BYTE 0] - start.byte) + 1
      TRUE
        message.size := size

    header [BYTE 1] := message.size/256 --- block.size (# of 256 bytes)
    header [BYTE 2] := message.size\256 ---              ( + remainder )
    header [BYTE 3] := channel.id   --- any tenth from 40 up to 240
    header [BYTE 4] := dest.transp  --- destination transputer
    out := route.table [dest.transp]
```

```
    BYTE.SLICE.OUTPUT (chan[channel.id + out],header,3,1) --- ready flag
    IF
      out = 4
        SEQ
          BYTE.SLICE.OUTPUT (link4,header,1,header.size)
          BYTE.SLICE.OUTPUT (link4,message,start.byte,message.size)
      out = 5
        SEQ
          BYTE.SLICE.OUTPUT (link5,header,1,header.size)
          BYTE.SLICE.OUTPUT (link5,message,start.byte,message.size)
      out = 6
        SEQ
          BYTE.SLICE.OUTPUT (link6,header,1,header.size)
          BYTE.SLICE.OUTPUT (link6,message,start.byte,message.size)
      out = 7
        SEQ
          BYTE.SLICE.OUTPUT (link7,header,1,header.size)
          BYTE.SLICE.OUTPUT (link7,message,start.byte,message.size)
    BYTE.SLICE.OUTPUT (chan[channel.id + out],header,3,1): --- done flag


-- PROC receive (VALUE channel.id,VAR message[], message.length[])
--- ***********************************************************
--- DESCRIPTION: It is an operating system routine, and it    *
--- is used to communicate between processors. It receives    *
--- the incoming message, and provides as an output parameter *
--- the size of the message just received. The parameter      *
--- channel id must have an exact match with the send         *
--- operation which originated that message.                  *
--- USAGE: receive (70,message.in,size)                       *
--- REMARK1: The user must be familiarized with the Operating *
--- System Structure before using this routine.               *
--- REMARK2: Notice that the message.length output parameter, *
--- must be a unity array of integers, while the message      *
--- itself must be declared as an array of bytes.             *
--- ***********************************************************

PROC receive (VALUE channel.id,VAR message[],message.length[])=
  SEQ
    WORD.SLICE.INPUT (chan[channel.id],message.length,0,1)
    BYTE.SLICE.INPUT (chan[channel.id],message,1,message.length[0]):


-- utilities
-- PROC tick.to.time (VALUE start, stop, board.type)
--- ***********************************************************
--- DESCRIPTION: It expects the board type which can be :     *
---   board.type = 0 ----> OPS (VAX VMS)                      *
---   board.type = 1 ----> B001 (T414:12.5 MHz)               *
---   board.type = 2 ----> B002                               *
---   board.type = 31----> B003 (T414:15 MHz - high pri)      *
---   board.type = 32----> B003 (T414:1 5MHz - low  pri)      *
---   board.type = 4 ----> B004                               *
--- and 2 signed integers representing some tick values       *
--- obtained by an assignment of the type TIME ? time.var     *
--- It then outputs the corrected elapsed time in hours, min,*
--- sec and msec, already taking into account the fact that   *
--- the timer wraps around when it reaches MAXINT or MININT.  *
--- USAGE: tickk.to.time (time1,time2,31)                     *
--- REMARK: Although it takes care of the wrapping, it won't  *
--- keep track of the number of times you have completed one  *
--- full cycle of the timer. In order to solve this problem   *
--- you should record roughly the start time. For example, in*
--- the VAX/VMS, the full cycle of the timer is 7.2 min, so   *
--- if you get the elapsed time of 5 min 7 sec 320 msec and   *
--- you have got a rough total time of 12 minutes, then the   *
--- real total time is 12 min 19 sec 320 msec.                *
--- ***********************************************************
```

```
PROC tick.to.time (VALUE start, stop, board.type) =
  -- constant definitions
  DEF vax.sec    =10000000 :        --- hundreds of nsec/second
  DEF vax.mili   =   10000 :        --- hundreds of nsec/milisecond
  DEF b001.sec   =  625000 :        --- # of 1.6 usec/second
  DEF b001.mili  =     625 :        --- # of 1.6 usec/milisecond
  DEF b003h.sec  = 1000000 :        --- # of usec/second
  DEF b003h.mili =    1000 :        --- # of usec/milisecond
  DEF b0031.sec  =   15625 :        --- # of 64 usec/second
  DEF b0031.mili =      16 :        --- # of 64 usec/milisecond

  DEF max.number.of.ticks = 2147483648 : --- maximum integer (2**31)
  VAR elapsed.tick :
  VAR factor1, factor2 :
  VAR msec, tot.sec, sec, min, hr :

  SEQ
    IF
      board.type = 0                          --- VAX VMS
        SEQ
          factor1 := vax.sec
          factor2 := vax.mili

      board.type  = 1                         --- B001
        SEQ
          factor1 := b001.sec
          factor2 := b001.mili

      board.type  = 2                         --- B002
        SKIP                                  --- not implemented

      board.type  = 31                        --- B003 in high priority
        SEQ
          factor1 := b003h.sec
          factor2 := b003h.mili

      board.type  = 32                        --- B003 in low priority
        SEQ
          factor1 := b0031.sec
          factor2 := b0031.mili

      board.type  = 4                         --- B004
        SKIP                                  --- not implemented

    elapsed.tick := stop - start
    IF
      elapsed.tick < 0
        elapsed.tick := elapsed.tick + max.number.of.ticks
      TRUE
        SKIP

    tot.sec := elapsed.tick/factor1
    hr      := tot.sec/3600
    min     := (tot.sec\3600)/60
    sec     := tot.sec\60
    msec    := (elapsed.tick\factor1)/factor2

    -- output time to screen
    write.number (hr)
    write.string (" hr ")
    write.number (min)
    write.string (" min ")
    write.number(sec)
    write.string (" sec ")
    write.number(msec)
    write.string (" msec") :


-- PROC dump (VALUE begin.address, count)
--- ************************************************************
```

```
--- DESCRIPTION: This procedure dumps the memory starting at *
--- the given "begin.address". The value for the             *
--- "begin.address" can be either in hex or decimal.         *
--- The count value determines how many words in memory will *
--- be retrieved.                                            *
--- USAGE:          a) dump (#80003540,100)                  *
---                 b) dump (1024,48)                        *
---                 c) dump (-5113,1024)                     *
--- REMARK1: When specifying the count value remember that   *
--- the retrieval is done by words, not bytes!!!             *
--- REMARK2: If count is not a multiple of 4 it will use the *
--- closest upper multiple.                                  *
--- REMARK3: Negatives or zero values for count although     *
--- accepted, will give you no output.                       *
--- ***********************************************************

PROC dump (VALUE begin.address, count) =
  VAR word.read:
  VAR hex.value [9], hex.addr[9]:
  VAR address, align, times:

  SEQ
    times := 0
    new.line(1)
    address := begin.address
    -- aligning a given address
    align := address\4
    IF
      align <> 0
        address := address - align
      TRUE
        SKIP


    WHILE times <= count
      SEQ
        write.string ("address ")
        dec.to.hex (address,hex.addr)
        write.string (hex.addr)
        write.string ("  -->  ")
        SEQ i = [0 FOR 4]
          SEQ
            GETWORD (word.read,address)
            dec.to.hex (word.read,hex.value)
            write.string (hex.value)
            space(2)
            times := times + 1
        SKIP
        address := address + 16
        new.line(1)
    SKIP:


-- PROC transfer.rate (VALUE start,stop,board.type,nr.of.bytes,VAR rate)
--- ***********************************************************
--- DESCRIPTION: It is basically the same routine as         *
--- tick.to.time, with the only difference that it returns a *
--- rate value in Kbits/sec instead of a time value.         *
--- USAGE: transfer.rate (time1,time2,31,4096,rate)          *
--- REMARK: If further information is needed, please refer to*
--- routine tick.to.time                                     *
--- ***********************************************************

PROC transfer.rate (VALUE start,stop,board.type,nr.of.bytes,VAR rate) =
  -- constant definitions
  DEF vax.sec    =10000000 :      --- hundreds of nsec/second
  DEF b001.sec   =  625000 :      --- # of 1.6 usec/second
  DEF b003h.sec  = 1000000 :      --- # of usec/second
  DEF b0031.sec  =   15625 :      --- # of 64 usec/second
  DEF max.number.of.ticks = 2147483648 : --- maximum integer (2**31)
```

```
  -- variable declarations
  VAR elapsed.tick :
  VAR factor :        --- to convert ticks to seconds

  SEQ
    elapsed.tick := stop - start
    IF
      elapsed.tick < 0
        elapsed.tick := elapsed.tick + max.number.of.ticks
      TRUE
        SKIP
    -- selection of correct factor iaw the board
    IF
      board.type = 0                          --- VAX VMS
        factor := vax.sec

      board.type  = 1                         --- B001
        factor := b001.sec

      board.type  = 2                         --- B002
        SKIP                                  --- not implemented

      board.type  = 31                        --- B003 in high priority
        factor := b003h.sec

      board.type  = 32                        --- B003 in low priority
        factor := b0031.sec

      board.type  = 4                         --- B004
        SKIP                                  --- not implemented

    -- rate calculation
    IF
      board.type = 32
        rate := ((nr.of.bytes*8)*factor)/(elapsed.tick*1000)
        --- operation is done this way to keep precision ok!
      TRUE
        rate := ((nr.of.bytes*8)*(factor/1000))/elapsed.tick
        --- operation is done this way in order not to exceed maxint
        --- on the numerator.
        --- multiply by 8 due to 8 bits per byte
        --- divide by 1000 to have the tranfer.rate in kbits/sec

    SKIP:

-- PROC capitalize (VAR ch[])
--- ***********************************************************
--- DESCRIPTION: It capitalizes the first character in any    *
--- string.                                                   *
--- USAGE: capitalize (string)                                *
--- ***********************************************************

PROC capitalize (VAR ch[]) =

  DEF delta =('a' - 'A') :
                            ---  A ---> 65
                            ---  a ---> 97      ASCII values
                            ---  z ---> 122

  SEQ
    IF
      (ch [BYTE 1] <= 'z') AND (ch [BYTE 1] >= 'a')
        ch [BYTE 1] := ch [BYTE 1] - delta
      TRUE
        SKIP :
```

```
-- PROC operating.system
PROC operating.system =
  -- PROC input.handler
  PROC input.handler =
    -- variable and constants declarations
    VAR header0 [BYTE 5],
        header1 [BYTE 5],
        header2 [BYTE 5],
        header3 [BYTE 5],

        buffer.in0 [BYTE max.block.size],
        buffer.in1 [BYTE max.block.size],
        buffer.in2 [BYTE max.block.size],
        buffer.in3 [BYTE max.block.size],

        block.size0[1], out0,
        block.size1[1], out1,
        block.size2[1], out2,
        block.size3[1], out3:

    SEQ
      -- initializing the buffers
      SEQ i = [0 FOR max.block.size]
        SEQ
          buffer.in0 [BYTE i] := '0'
          buffer.in1 [BYTE i] := '1'
          buffer.in2 [BYTE i] := '2'
          buffer.in3 [BYTE i] := '3'
      SKIP

      PAR
        WHILE TRUE
          -- listen to link0
          SEQ
            -- receiving the header
            BYTE.SLICE.INPUT (link0,header0,1,header.size)

            -- decoding the block size
            block.size0[0] := ((256 * header0[BYTE 1])+header0[BYTE 2])

            -- buffering the message
            BYTE.SLICE.INPUT (link0,buffer.in0,1,block.size0[0])

            IF
              -- the message is to be bypassed
              header0 [BYTE 4] <> this.transputer
                SEQ
                  -- finding the best link to output that message
                  out0 := route.table [header0 [BYTE 4]]

                  -- outputing to the required link
                  --- request flag thru chan 4, 5, 6 or 7
                  BYTE.SLICE.OUTPUT(chan[out0],header0,3,1)
                  IF
                    out0 = 4
                      SEQ
                        BYTE.SLICE.OUTPUT (link4,header0,1,header.size)
                        BYTE.SLICE.OUTPUT (link4,buffer.in0,1,
                                           block.size0[0])
                    out0 = 5
                      SEQ
                        BYTE.SLICE.OUTPUT (link5,header0,1,header.size)
                        BYTE.SLICE.OUTPUT (link5,buffer.in0,1,
                                           block.size0[0])
                    out0 = 6
                      SEQ
                        BYTE.SLICE.OUTPUT (link6,header0,1,header.size)
                        BYTE.SLICE.OUTPUT (link6,buffer.in0,1,
                                           block.size0[0])
                    out0 = 7
```

```
                SEQ
                  BYTE.SLICE.OUTPUT (link7,header0,1,header.size)
                  BYTE.SLICE.OUTPUT (link7,buffer.in0,1,
                                                block.size0[0])
            --- release flag
            BYTE.SLICE.OUTPUT(chan[out0],header0,3,1)

        -- the message is for this transputer
        header0 [BYTE 4] = this.transputer
          SEQ
            IF
              header0 [BYTE 3] <> scrn
                SEQ
                  -- passing the size of the message
                                                (block.size0[0])
                  WORD.SLICE.OUTPUT (chan[header0 [BYTE 3]],
                                                block.size0,0,1)

                  -- passing the message itself
                  BYTE.SLICE.OUTPUT (chan[header0 [BYTE 3]],
                                                buffer.in0,1,block.size0[0])

              TRUE      --- if channel.id = 40 = scrn
                SEQ
                  -- I'm ready
                  BYTE.SLICE.OUTPUT (screen[0],header0,3,1)

                  -- outputting to the screen
                  send.string (Screen,buffer.in0,1,block.size0[0])

                  -- I'm done
                  BYTE.SLICE.OUTPUT (screen[0],header0,3,1)

WHILE TRUE
  -- listen to link1
  SEQ
    -- receiving the header
    BYTE.SLICE.INPUT (link1,header1,1,header.size)

    -- decoding the block size
    block.size1[0] := ((256 * header1[BYTE 1])+header1[BYTE 2])

    -- buffering the message
    BYTE.SLICE.INPUT (link1,buffer.in1,1,block.size1[0])

    IF
      -- the message is to be bypassed
      header1 [BYTE 4] <> this.transputer
        SEQ
          -- finding the best link to output that message
          out1 := route.table [header1 [BYTE 4]]

          -- outputing to the required link
          --- request flag thru chan 14, 15, 16 or 17
          BYTE.SLICE.OUTPUT(chan[10+out1],header1,3,1)
          IF
            out1 = 4
              SEQ
                BYTE.SLICE.OUTPUT (link4,header1,1,header.size)
                BYTE.SLICE.OUTPUT (link4,buffer.in1,1,
                                                block.size1[0])
            out1 = 5
              SEQ
                BYTE.SLICE.OUTPUT (link5,header1,1,header.size)
                BYTE.SLICE.OUTPUT (link5,buffer.in1,1,
                                                block.size1[0])
            out1 = 6
              SEQ
                BYTE.SLICE.OUTPUT (link6,header1,1,header.size)
                BYTE.SLICE.OUTPUT (link6,buffer.in1,1,
```

```
                                        block.size1[0])
              out1 = 7
                SEQ
                  BYTE.SLICE.OUTPUT (link7,header1,1,header.size)
                  BYTE.SLICE.OUTPUT (link7,buffer.in1,1,
                                        block.size1[0])
            --- release flag
            BYTE.SLICE.OUTPUT(chan[10+out1],header1,3,1)

        -- the message is for this transputer
        header1 [BYTE 4] = this.transputer
          SEQ
            IF
              header1 [BYTE 3] <> scrn
                SEQ
                  -- passing the size of the message
                                        (block.size1[0])
                  WORD.SLICE.OUTPUT (chan[header1 [BYTE 3]],
                                        block.size1,0,1)

                  -- passing the message itself
                  BYTE.SLICE.OUTPUT (chan[header1 [BYTE 3]],
                                        buffer.in1,1,block.size1[0])

              TRUE      --- if channel.id = 40 = scrn
                SEQ
                  -- I'm ready
                  BYTE.SLICE.OUTPUT (screen[1],header1,3,1)

                  -- outputting to the screen
                  send.string (Screen,buffer.in1,1,block.size1[0])

                  -- I'm done
                  BYTE.SLICE.OUTPUT (screen[1],header1,3,1)

WHILE TRUE
  -- listen to link2
  SEQ
    -- receiving the header
    BYTE.SLICE.INPUT (link2,header2,1,header.size)

    -- decoding the block size
    block.size2[0] := ((256 * header2[BYTE 1])+header2[BYTE 2])

    -- buffering the message
    BYTE.SLICE.INPUT (link2,buffer.in2,1,block.size2[0])

    IF
      -- the message is to be bypassed
      header2 [BYTE 4] <> this.transputer
        SEQ
          -- finding the best link to output that message
          out2 := route.table [header2 [BYTE 4]]

          -- outputing to the required link
          --- request flag thru chan 24, 25, 26 or 27
          BYTE.SLICE.OUTPUT(chan[20+out2],header2,3,1)
          IF
            out2 = 4
              SEQ
                BYTE.SLICE.OUTPUT (link4,header2,1,header.size)
                BYTE.SLICE.OUTPUT (link4,buffer.in2,1,
                                        block.size2[0])
            out2 = 5
              SEQ
                BYTE.SLICE.OUTPUT (link5,header2,1,header.size)
                BYTE.SLICE.OUTPUT (link5,buffer.in2,1,
                                        block.size2[0])
            out2 = 6
              SEQ
```

```
                BYTE.SLICE.OUTPUT (link6,header2,1,header.size)
                BYTE.SLICE.OUTPUT (link6,buffer.in2,1,
                                   block.size2[0])
            out2 = 7
              SEQ
                BYTE.SLICE.OUTPUT (link7,header2,1,header.size)
                BYTE.SLICE.OUTPUT (link7,buffer.in2,1,
                                   block.size2[0])
          --- release flag
          BYTE.SLICE.OUTPUT(chan[20+out2],header2,3,1)

      -- the message is for this transputer
      header2 [BYTE 4] = this.transputer
        SEQ
          IF
            header2 [BYTE 3] <> scrn
              SEQ
                -- passing the size of the message
                                   (block.size2[0])
                WORD.SLICE.OUTPUT (chan[header2 [BYTE 3]],
                                   block.size2,0,1)

                -- passing the message itself
                BYTE.SLICE.OUTPUT (chan[header2 [BYTE 3]],
                                   buffer.in2,1,block.size2[0])

            TRUE      --- if channel.id = 40 = scrn
              SEQ
                -- I'm ready
                BYTE.SLICE.OUTPUT (screen[2],header2,3,1)

                -- outputting to the screen
                send.string (Screen,buffer.in2,1,block.size2[0])

                -- I'm done
                BYTE.SLICE.OUTPUT (screen[2],header2,3,1)

WHILE TRUE
  -- listen to link3
  SEQ
    -- receiving the header
    BYTE.SLICE.INPUT (link3,header3,1,header.size)

    -- decoding the block size
    block.size3[0] := ((256 * header3[BYTE 1])+header3[BYTE 2])

    -- buffering the message
    BYTE.SLICE.INPUT (link3,buffer.in3,1,block.size3[0])

    IF
      -- the message is to be bypassed
      header3 [BYTE 4] <> this.transputer
        SEQ
          -- finding the best link to output that message
          out3 := route.table [header3 [BYTE 4]]

          -- outputing to the required link
          --- request flag thru chan 34, 35, 36 or 37
          BYTE.SLICE.OUTPUT(chan[30+out3],header3,3,1)
          IF
            out3 = 4
              SEQ
                BYTE.SLICE.OUTPUT (link4,header3,1,header.size)
                BYTE.SLICE.OUTPUT (link4,buffer.in3,1,
                                   block.size3[0])
            out3 = 5
              SEQ
                BYTE.SLICE.OUTPUT (link5,header3,1,header.size)
                BYTE.SLICE.OUTPUT (link5,buffer.in3,1,
                                   block.size3[0])
```

```
                    out3 = 6
                      SEQ
                        BYTE.SLICE.OUTPUT (link6,header3,1,header.size)
                        BYTE.SLICE.OUTPUT (link6,buffer.in3,1,
                                                  block.size3[0])
                    out3 = 7
                      SEQ
                        BYTE.SLICE.OUTPUT (link7,header3,1,header.size)
                        BYTE.SLICE.OUTPUT (link7,buffer.in3,1,
                                                  block.size3[0])
                  --- release flag
                  BYTE.SLICE.OUTPUT(chan[30+out3],header3,3,1)

                -- the message is for this transputer
                header3 [BYTE 4] = this.transputer
                  SEQ
                    IF
                      header3 [BYTE 3] <> scrn
                        SEQ
                          -- passing the size of the message
                                                  (block.size3[0])
                          WORD.SLICE.OUTPUT (chan[header3 [BYTE 3]],
                                             block.size3,0,1)

                          -- passing the message itself
                          BYTE.SLICE.OUTPUT (chan[header3 [BYTE 3]],
                                             buffer.in3,1,block.size3[0])

                      TRUE      --- if channel.id = 40 = scrn
                        SEQ
                          -- I'm ready
                          BYTE.SLICE.OUTPUT (screen[3],header3,3,1)

                          -- outputting to the screen
                          send.string (Screen,buffer.in3,1,block.size3[0])

                          -- I'm done
                          BYTE.SLICE.OUTPUT (screen[3],header3,3,1) :

-- PROC output.handler
PROC output.handler =
  -- local variable declarations
  VAR flag4 [BYTE 2]:
  VAR flag5 [BYTE 2]:
  VAR flag6 [BYTE 2]:
  VAR flag7 [BYTE 2]:

  PAR
    WHILE TRUE
      ALT i = [0 FOR max.io.channels]
        chan [(10*i) +4] ? flag4 [BYTE 0] --- for link4
          BYTE.SLICE.INPUT (chan [(10*i) +4],flag4,0,1)
    WHILE TRUE
      ALT j = [0 FOR max.io.channels]
        chan [(10*j) +5] ? flag5 [BYTE 0] --- for link5
          BYTE.SLICE.INPUT (chan [(10*j) +5],flag5,0,1)
    WHILE TRUE
      ALT k = [0 FOR max.io.channels]
        chan [(10*k) +6] ? flag6 [BYTE 0] --- for link6
          BYTE.SLICE.INPUT (chan [(10*k) +6],flag6,0,1)
    WHILE TRUE
      ALT l = [0 FOR max.io.channels]
        chan [(10*l) +7] ? flag7 [BYTE 0] --- for link7
          BYTE.SLICE.INPUT (chan [(10*l) +7],flag7,0,1):


-- PROC screen.handler
PROC screen.handler =
  VAR flag [BYTE 2]:
```

```
  WHILE TRUE
    ALT i = [0 FOR max.screen.channels]
      screen[i] ? flag[BYTE 1]
        BYTE.SLICE.INPUT (screen[i],flag,1,1):


-- SC PROC terminal.driver(CHAN Keyboard,Screen,VALUE port,baud.rate)
--- ********************************************************
--- This routine is provided by the manufacturer, and it   *
--- varies with the board we are using. This particular    *
--- one is for the B001 board.                             *
--- ********************************************************

-- PROC terminal.driver(CHAN Keyboard,Screen,VALUE port,baud.rate)
PROC terminal.driver (CHAN Keyboard, Screen, VALUE port, baud.rate)=
  -- T414 Board Definitions
  -- declare constants
  DEF  bpw            = 4 :
  DEF  bits.per.word  = 32 :
  DEF  perif.base     = #80040000 :  -- base address of peripherals

  -- duart register addresses

  -- See table 1 'Register addressing' on page 6 of
  --                        the SCN2681 data sheet.
  -- These are all word offsets from address zero

  DEF uartA  = perif.base + 0 :
  DEF uartB  = perif.base + (8 * bpw) :

  DEF mode.reg                = 0 * bpw :     --            MR

  DEF status.reg              = 1 * bpw :     -- read       SR
  DEF clock.select.reg        = 1 * bpw :     -- write      CSR

  DEF command.reg             = 2 * bpw :     --            CR

  DEF rx.reg                  = 3 * bpw :     -- read
  DEF tx.reg                  = 3 * bpw :     -- write

  DEF input.port.change.reg   = 4 * bpw :     -- read  IPCR uartA only
  DEF aux.control.reg         = 4 * bpw :     -- write ACR  uartA only

  DEF interrupt.status.reg    = 5 * bpw :     -- read  ISR  uartA only
  DEF interrupt.mask.reg      = 5 * bpw :     -- write IMR  uartA only

  DEF input.port              = 5 * bpw :     -- read       uartB only
  DEF output.port.conf.reg    = 5 * bpw :     -- write OPCR uartB only

  DEF timer.upper.reg         = 6 * bpw :     --       CTU  uartA only
  DEF timer.lower.reg         = 7 * bpw :     --       CTL  uartA only

  DEF start.counter           = 6 * bpw :     -- read       uartB only
  DEF set.output.port.bits    = 6 * bpw :     -- write      uartB only

  DEF stop.counter            = 7 * bpw :     -- read       uartB only
  DEF reset.output.port.bits  = 7 * bpw :     -- write      uartB only

  -- declare register values
  -- MR1  mode register 1

  DEF rx.rts.control      = #00 :   -- [7]    no rts control
  DEF rx.int.select       = #00 :   -- [6]    interrupt on rx.ready
  DEF error.mode          = #00 :   -- [5]    character error mode
  DEF parity.mode         = #10 :   -- [4:3]  disable parity
  DEF parity.type         = #00 :   -- [2]    even parity
  DEF bits.per.char       = #03 :   -- [1:0]  8 bits per char

  DEF MR1.control = rx.rts.control \/
                    rx.int.select  \/
```

```
                    error.mode      \/
                    parity.mode     \/
                    parity.type     \/
                    bits.per.char   :

-- MR2  mode register 2

DEF channel.mode     = #00 :     -- [7:6]  normal channel mode
DEF tx.trs.control   = #00 :     -- [5]    rts control not used
DEF cts.enable.tx    = #00 :     -- [4]    cts control not used
DEF stop.bit.length  = #07 :     -- [3:0]  1.000 stop bits

DEF MR2.control = channel.mode     \/
                  tx.trs.control   \/
                  cts.enable.tx    \/
                  stop.bit.length  :

-- CR   command register


DEF bit.seven           = #00 : -- [7] not used must be zero

                                -- [6:4] misc comds never combined

DEF no.command          = #00 :
DEF reset.mr.ptr        = #10 : -- make mode register point at MR1
DEF reset.rx            = #20 :
DEF reset.tx            = #30 :
DEF reset.error         = #40 :
DEF reset.break         = #50 :
DEF start.break         = #60 :
DEF stop.break          = #70 :

DEF enable.rx           = #01 :  -- [3]
DEF disable.rx          = #02 :  -- [2]
DEF enable.tx           = #04 :  -- [1]
DEF disable.tx          = #08 :  -- [0]

-- SR   status register

DEF received.break = #80 :  -- [7]
DEF framing.error  = #40 :  -- [6]
DEF parity.error   = #20 :  -- [5]
DEF overrun.error  = #10 :  -- [4]
DEF tx.empty       = #08 :  -- [3]
DEF tx.ready       = #04 :  -- [2]
DEF fifo.full      = #02 :  -- [1]
DEF rx.ready       = #01 :  -- [0]

-- OPCR output port configuration register

-- Mask this beast out before programming the timer

DEF OPCR.control = #00 :  -- [7:0] mask out output port

-- ACR  aux control register


DEF brg.set.select       = #00 : -- [7]    select set 1 baud rates
                                 --        for CSRA
DEF counter.timer.mode   = #00 : -- [6:4]  external counter
DEF delta.ip3.0.int      = #00 : -- [3:0]  no bits in IPCR affect
                                 --        in IMR [7]

DEF ACR.control = brg.set.select     \/
                  counter.timer.mode \/
                  delta.ip3.0.int    :

-- IMR  interrupt mask register
```

```
DEF IMR.control = #00:    -- [7:0] no interrupts

-- PAL bit registers

-- RS232 RX data and switches

-- T414 i/o procs
-- PROC reset.uart (VALUE uart,baud.rate)
PROC reset.uart (VALUE uart, baud.rate)=
  VAR now, the.future :
  SEQ
    PUTBYTE (reset.mr.ptr\/disable.rx /disable.tx,uart+command.reg)
    PUTBYTE (MR1.control, uart + mode.reg)
    PUTBYTE (MR2.control, uart + mode.reg)
    PUTBYTE (ACR.control, uartA + aux.control.reg)
    PUTBYTE (baud.rate, uart + clock.select.reg)
    PUTBYTE (no.command\/enable.rx /enable.tx,uart+command.reg)
    -- wait a bit
    TIME ? the.future
    TIME ? now
    the.future := the.future + #40000
    WHILE the.future AFTER now
      TIME ? now

    SKIP:

-- PROC read (CHAN out, VALUE uart)
PROC read (CHAN out, VALUE uart) =
  -- read from keyboard with deschedule between polls
  VAR  status, ch :
  SEQ
    WHILE TRUE
      SEQ
        -- read status
        GETBYTE (status, uart + status.reg)

        -- wait for received character
        WHILE (status /\ rx.ready) = 0
          PAR
            SKIP
            -- try status again
            GETBYTE (status, uart + status.reg)

        -- read the character
        GETBYTE (ch, uart + rx.reg)

        -- output the character
        out ! ch

    SKIP :

-- PROC write (CHAN in,  VALUE uart)
PROC write (CHAN in,  VALUE uart) =
  -- write to uart
  VAR uart.failed :
  SEQ
    uart.failed := FALSE
    WHILE TRUE
      VAR  ch :
      SEQ
        in ? ch
        IF
          (ch < 0) OR (uart.failed)
            SKIP
          TRUE
            -- wrch (VALUE ch, uart) with timeout
            DEF timeout =  3200000 :
            VAR status, count :
            SEQ
              status := 0
```

```
                    count  := 0
                    WHILE ((status /\ tx.ready) = 0) AND (count < timeout)
                      SEQ
                        GETBYTE (status, uart + status.reg)
                        count := count + 1
                    IF
                      count = timeout
                        uart.failed := FALSE   ---TRUE
                      TRUE
                        PUTBYTE (ch, uart + tx.reg)

        SKIP :

-- main program
VAR uart :
SEQ
  IF
    port = 0
      uart := uartA
    TRUE
      uart := uartB
  reset.uart (uart, baud.rate \/ (baud.rate << 4))
  PAR
    read  (Keyboard, uart)
    write (Screen,   uart)
  SKIP :
```

```
-- main body of the operating system
SEQ
  -- receiving the routing table
  route.table[0]   := t0          --- output link to transp #0
  route.table[1]   := t1          --- output link to transp #1
  route.table[2]   := t2          --- output link to transp #2
  route.table[3]   := t3          --- output link to transp #3
  route.table[4]   := t4          --- output link to transp #4
  route.table[5]   := t5          --- output link to transp #5
  route.table[6]   := t6          --- output link to transp #6
  route.table[7]   := t7          --- output link to transp #7
  route.table[8]   := t8          --- output link to transp #8
  route.table[9]   := t9          --- output link to transp #9
  route.table[10]  := t10         --- output link to transp #10
  route.table[11]  := t11         --- output link to transp #11
  route.table[12]  := t12         --- output link to transp #12
  route.table[13]  := t13         --- output link to transp #13
  route.table[14]  := t14         --- output link to transp #14
  route.table[15]  := t15         --- output link to transp #15
  route.table[16]  := t16         --- output link to transp #16
  route.table[17]  := t17         --- output link to transp #17

  PAR
    output.handler
    input.handler
    terminal.driver(Keyboard,Screen,port,baud)
    screen.handler :
```

APPENDIX E

THE OPERATING SYSTEM FOR REMOTE TRANSPUTERS (REMOTE_OS.TDS)

```
--- ***********************************************************
--- *   Title:  REMOTE_OS.TDS                  * Version: 1.0  *
--- *   Author: MAURICIO DE MENEZES CORDEIRO * Mod: 0        *
--- *   Date:   13/MAI/1987                    *****************
--- *   Programming Language:  OCCAM 1                        *
--- *   Compiler:  IMS D-600 (VAX/VMS)                        *
--- *   Brief Description: This program contains the source   *
--- *   code for a communications operating system for remote*
--- *   processors in a network of transputers. It must be    *
--- *   placed in parallel with the user process.             *
--- ***********************************************************
--- *   Mod #:       Date:                                    *
--- *   Responsible:                                          *
--- *   Brief Description:                                    *
--- *                                                         *
--- ***********************************************************
--- *   Mod #:       Date:                                    *
--- *   Responsible:                                          *
--- *   Brief Description:                                    *
--- *                                                         *
--- ***********************************************************
-- Operating System global declarations
DEF max.block.size = 4100:
DEF nr.of.transputers = 17:
DEF header.size = 4:
DEF scrn = 40:                          --- channel screen
DEF max.io.channels = 25:               --- 0 up to 240, in tenths
DEF max.screen.channels = 5:
VAR route.table[18]:
VAR flag [BYTE 1]:                      --- for the library routines
CHAN chan [10 * max.io.channels]:       --- Actually it should be :
                                        ---(10*(max.io.channels-1))+8
CHAN screen [max.screen.channels]:


-- global_def.tds
--- ***********************************************************
---      At this point we should imbed the filed fold         *
---    global_def.tds, which is described in Appendix B       *
--- ***********************************************************


-- Operating System Channel Placements
CHAN link0 AT link0in :
CHAN link1 AT link1in :
CHAN link2 AT link2in :
CHAN link3 AT link3in :
CHAN link4 AT link0out:
CHAN link5 AT link1out:
CHAN link6 AT link2out:
CHAN link7 AT link3out:


-- remote_lib.tds
-- io_routines
-- PROC dec.to.hex (VALUE integer, VAR string[])
--- ***********************************************************
--- DESCRIPTION: It converts an integer number from its       *
--- decimal representation into the equivalent hexadecimal    *
--- one. It accepts any valid integer. It returns the         *
```

```
--- hexadecimal number stored in a string of 10 bytes long   *
--- where the leading zeros are preserved.                   *
--- It returns the following format: [size]#0000FFFF         *
--- USAGE: dec.to.hex(37182,hex.string)                      *
--- REMARK: The BYTE[0] of the string carries its length     *
--- which is always 9, therefore it could be deleted, but    *
--- we decided to keep it.                                   *
--- ******************************************************** *

PROC dec.to.hex (VALUE integer, VAR string []) =
  VAR first, order.of.digit, digit :
  VAR number :
  DEF hex.char = "0123456789ABCDEF" :
  SEQ
    first := TRUE
    string [BYTE 0] := 9
    string [BYTE 1] := '#'
    number := integer
    order.of.digit := 9
    WHILE (number > 0) OR (first=TRUE)
      SEQ
        digit := number /\ #F
        digit := hex.char [BYTE digit + 1]
        string [BYTE order.of.digit] := digit
        number := number >> 4
        order.of.digit := order.of.digit - 1
        first := FALSE
    SEQ i = [2 FOR (order.of.digit - 1)]
      string [BYTE i] := '0':


-- PROC dec.to.ascii (VALUE integer, VAR string [])
--- ********************************************************
--- DESCRIPTION: It converts an integer number from its      *
--- decimal representation into the equivalent ASCII one. It *
--- accepts any valid integer number. It returns the ASCII   *
--- number stored in a string of 12 bytes long, where the    *
--- number is right justified and it has the following       *
--- format:  - 3542  ---> '-' ' ' ' ' ' ' ' ' '3''5''4''2'  *
---          1922937  ---> ' ' ' ' ' ' '1''9''2''2''9''3''7' *
--- USAGE: dec.to.ascii(-9873,ascii.string)                  *
--- REMARK: The BYTE[0] of the string carries its length     *
--- which is always 11, therefore it could be eliminated, but*
--- we decided to keep it.                                   *
--- ********************************************************

PROC dec.to.ascii (VALUE integer, VAR string []) =
  VAR number :
  VAR order.of.digit :
  DEF min.int = - 2147483648 :

  SEQ
    number := integer
    order.of.digit := 11
    string [BYTE 0] := 11
    IF
      number = min.int
        -- taking care of the limit case
        SEQ
          string [BYTE 1]  := '-'
          string [BYTE 2]  := '2'
          string [BYTE 3]  := '1'
          string [BYTE 4]  := '4'
          string [BYTE 5]  := '7'
          string [BYTE 6]  := '4'
          string [BYTE 7]  := '8'
          string [BYTE 8]  := '3'
          string [BYTE 9]  := '6'
          string [BYTE 10] := '4'
          string [BYTE 11] := '8'
```

```
      TRUE
        SEQ
          IF
            number = 0
              SEQ
                string [BYTE 1] := ' '
                string [BYTE 11]:= '0'
                order.of.digit := 10
            number < 0
              SEQ
                number := - number
                string [BYTE 1] := '-'
            TRUE                                --- number > 0
              string [BYTE 1] := ' '
          -- building up the actual number
          WHILE number > 0
            SEQ
              string [BYTE order.of.digit] := (number \ 10) + '0'
              number := (number / 10)
              order.of.digit := order.of.digit - 1

          SEQ i = [2 FOR (order.of.digit - 1)]
            string [BYTE i] := ' ':


-- PROC hex.to.dec (VALUE string[], VAR integer, OK)
--- ************************************************************
--- DESCRIPTION: It accepts a hexadecimal representation of  *
---  a number and converts it into an integer number. It     *
--- expects the byte[0] of the string to carry the size      *
--- information of that "hex number".                        *
--- USAGE: hex.to.dec ("#00003785",number,valid)             *
---        hex.to.dec ("#1452",number,valid)                 *
---        hex.to.dec ("#19574",number,valid)                *
---        ascii.to.dec (hex.string,number,valid)            *
--- REMARK: Returns a boolean value FALSE in OK if the       *
--- string is not in the correct format.                     *
--- ************************************************************

PROC hex.to.dec (VALUE string [], VAR integer, OK) =
  SEQ
    integer := 0
    IF
      -- empty string
      string [BYTE 0] = 0
        OK := FALSE

      -- hex number
      string [BYTE 0] <> 0
        IF
          -- starts with '#'
          string [BYTE 1] = '#'
            VAR Count :
            SEQ
              OK := TRUE
              Count := 2
              WHILE (Count <= string [BYTE 0]) AND OK
                VAR Digit :
                SEQ
                  DEF hexChars = "0123456789ABCDEF" :
                  IF
                    IF Index = [1 FOR hexChars [BYTE 0]]
                      hexChars [BYTE Index] = string [BYTE Count]
                        Digit := Index - 1
                    TRUE
                      OK := FALSE
                  integer := (integer << 4) + Digit
                  Count := Count + 1
```

```
          -- otherwise
          string [BYTE 1] <> '#'
            OK := FALSE
    SKIP :


-- PROC ascii.to.dec (VALUE string[], VAR integer, OK)
--- ***********************************************************
--- DESCRIPTION: It accepts an ascii decimal                    *
--- representation of a number and converts it into an          *
--- integer number. It expects the byte[0] of the string        *
--- to carry the size information of that "ascii number".       *
--- USAGE: ascii.to.dec ("-3785",number,valid)                  *
---        ascii.to.dec ("+1452",number,valid)                  *
---        ascii.to.dec ("19574",number,valid)                  *
---        ascii.to.dec (string,number,valid)                   *
--- REMARK: Returns a boolean value FALSE in OK if the          *
--- string is not in the correct format.                        *
--- ***********************************************************

PROC ascii.to.dec (VALUE string [], VAR integer, OK) =
  SEQ
    integer := 0
    IF
      -- empty string
      string [BYTE 0] = 0
        OK := FALSE

      -- number
      string [BYTE 0] <> 0
        VAR Sign :
        VAR Start :
        VAR Length :
        SEQ
          OK := TRUE
          IF
            -- negative
            string [BYTE 1] = '-'
              SEQ
                Sign := - 1
                Start := 2
                Length := string [BYTE 0] - 1

            -- positive
            string [BYTE 1] <> '-'
              SEQ
                Sign := 1
                Start := 1
                Length := string [BYTE 0]

          -- convert to integer
          SEQ Index = [Start FOR Length]
            VAR Digit :
            SEQ
              Digit := string [BYTE Index]
              IF
                ('0' <= Digit) AND (Digit <= '9')
                  integer := (integer * 10) + (Digit - '0')
                TRUE
                  OK := FALSE

          integer := integer * Sign
    SKIP :


-- PROC rem.write.number (VALUE integer, root.number)
--- ***********************************************************
--- DESCRIPTION: This PROC outputs a signed integer value to *
--- the screen. It left justifies the number, so that if you *
--- need it right justified, use the dec.to.ascii and then   *
```

```
--- the write.string routines.                                          *
--- It uses the following format:                                       *
---                        0    --->  0                                *
---                  -234193    --->  -234193                          *
---                     1496    --->  149                              *
--- USAGE: write.number (integer)                                       *
---        write.number(135)                                            *
--- REMARK:  IT IS TO BE USED JUST IN REMOTE TRANSPUTERS                *
--- ********************************************************************

PROC rem.write.number(VALUE integer, root.number) =
  DEF min.int = - 2147483648 :
  DEF max.digits = 11 :
  VAR number, j:
  VAR order.of.digit :
  VAR digit [BYTE 12] :
  VAR string [BYTE 12] :

  SEQ
    j := 1
    number := integer
    order.of.digit := 11
    IF
      number = 0
        send (40,root.number,"0 ",1,1)
      number = min.int
        send (40,root.number,"-2147483648",1,11)
      TRUE
        SEQ
          IF
            number < 0
              SEQ
                number := - number
                string [BYTE 0] := '-'
            TRUE                                --- number > 0
              string [BYTE 0] := ' '
          WHILE number > 0
            SEQ
              digit [BYTE order.of.digit] := (number \ 10) + '0'
              number := (number / 10)
              order.of.digit := order.of.digit - 1
          SEQ i = [(order.of.digit+1) FOR (max.digits-order.of.digit)]
            SEQ
              string [BYTE j] := digit [BYTE i]
              j := j+1
          send (40,root.number,string,0,j) :


-- PROC rem.clear.screen (VALUE root.number)
--- ********************************************************************
--- DESCRIPTION: It clears the screen and homes the cursor.             *
--- USAGE: clear.screen                                                 *
--- REMARK: IT IS TO BE USED JUST IN REMOTE TRANSPUTERS                 *
--- ********************************************************************

PROC rem.clear.screen (VALUE root.number) =
  VAR string [BYTE 7] :
  SEQ
    string [BYTE 0] := esc          --- clear screen sequence
    string [BYTE 1] := '['
    string [BYTE 2] := '2'
    string [BYTE 3] := 'J'

    string [BYTE 4] := esc          --- home cursor
    string [BYTE 5] := '['
    string [BYTE 6] := 'H'

    send (40,root.number,string,0,7):
```

```
-- PROC rem.pos.cursor (VALUE line, column, root.number)
--- ***********************************************************
--- DESCRIPTION: Positions the cursor in a specified line and*
--- column. We have used the ANSI escape sequence             *
--- ESC [Line;Column H.                                       *
--- USAGE: pos.cursor (8,30)                                  *
--- REMARK1: Valid values for line   are 0 up to 24           *
---          Valid values for column are 0 up to 80           *
--- REMARK2: Values out of the above range will cause         *
---          unpredictable results.                           *
--- REMARK3: IT IS TO BE USED JUST IN REMOTE TRANSPUTERS      *
--- ***********************************************************

PROC rem.pos.cursor (VALUE line, column, root.number) =
  VAR string [BYTE 8]:
  VAR x [BYTE 2], y [BYTE 2]:
  SEQ
    IF
      (line < 10) AND (line >= 0)
        SEQ
          y [BYTE 0] := '0'
          y [BYTE 1] := line + #30
      (line >= 10) AND (line <= 24)
        SEQ
          y [BYTE 0] := (line/10) + #30
          y [BYTE 1] := (line\10) + #30
      TRUE
        SKIP
    IF
      (column < 10) AND (column >= 0)
        SEQ
          x [BYTE 0] := '0'
          x [BYTE 1] := column + #30
      (column >= 10) AND (column <= 80)
        SEQ
          x [BYTE 0] := (column/10) + #30
          x [BYTE 1] := (column\10) + #30
      TRUE
        SKIP
    string [BYTE 0] := esc
    string [BYTE 1] := '['
    string [BYTE 2] := y [BYTE 0]
    string [BYTE 3] := y [BYTE 1]
    string [BYTE 4] := ';'
    string [BYTE 5] := x [BYTE 0]
    string [BYTE 6] := x [BYTE 1]
    string [BYTE 7] := 'H'
    send (40,root.number,string,0,8):


-- PROC rem.new.line (VALUE number, root.number)
--- ***********************************************************
--- DESCRIPTION: It will skip as many lines as specified in   *
--- its parameters list.                                      *
--- USAGE: new.line(4)                                        *
--- REMARK1: Negative numbers will not give any new lines.    *
--- REMARK2: IT IS TO BE USED JUST IN REMOTE TRANSPUTERS      *
--- ***********************************************************

PROC rem.new.line (VALUE number, root.number) =
  VAR string [BYTE 2]:
  SEQ
    string[ BYTE 0] := cr
    string[ BYTE 1] := lf
    SEQ i = [0 FOR number]
      send (40,root.number,string,0,2):


-- PROC rem.space (VALUE number, root.number)
--- ***********************************************************
```

```
--- DESCRIPTION This procedure provides spaces for formatting*
--- a single line.                                            *
--- USAGE: space(8)                                           *
--- REMARK1: This routine does not provide an automatic lf    *
--- after reaching the end of the line.                       *
--- REMARK2: IT IS TO BE USED JUST IN REMOTE TRANSPUTERS      *
--- ***********************************************************

PROC rem.space (VALUE number, root.number) =
  VAR string [BYTE 1]:
  SEQ
    string [BYTE 0] := sp
    SEQ i = [0 FOR number]
      send (40,root.number,string,0,1):


-- PROC rem.tab (VALUE number, root.number)
--- ***********************************************************
--- DESCRIPTION This procedure provides tabs for formatting a*
--- single line. Each tab is equivalent to 8 spaces if the    *
--- terminal is using the default set up.                     *
--- USAGE: tab(6)                                             *
--- REMARK1: This routine does not provide an automatic lf    *
--- after reaching the end of the line.                       *
--- REMARK2: IT IS TO BE USED JUST IN REMOTE TRANSPUTERS      *
--- ***********************************************************

PROC rem.tab (VALUE number, root.number) =
  VAR string [BYTE 1]:
  SEQ
    string [BYTE 0] := tab
    SEQ i = [0 FOR number]
      send (40,root.number,string,0,1):


-- PROC send(VALUE channel.id,dest.transp,message[],start.byte,size)
--- ***********************************************************
--- DESCRIPTION: It is an operating system routine, and it     *
--- is used to communicate between processors. It builds the   *
--- header of the message to be sent. It has as parameters     *
--- the channel id of the channel which is going to carry on   *
--- the communications, the id of the destination transputer   *
--- for that message, the start byte and the size of the       *
--- message to be transmitted. For every send must exist a     *
--- receive for that same channel id in the destination        *
--- transputer.                                               *
--- USAGE: send (70,4,message,1,0)                            *
--- REMARK: The user must be familiarized with the Operating   *
--- System Structure before using this routine.               *
--- ***********************************************************

PROC send (VALUE channel.id,dest.transp,message[],start.byte,size)=
  VAR out,message.size,header [BYTE 5]:
  SEQ
    IF
      size <= 0  --- send from the start.byte all way to the end.
                 --- this method is valid for messages up to 255 bytes.
                 --- even for size < 0 it behaves like it was a 0.
        message.size := (message[BYTE 0] - start.byte) + 1
      TRUE
        message.size := size

    header [BYTE 1] := message.size/256 --- block.size (# of 256 bytes)
    header [BYTE 2] := message.size\256 ---              ( + remainder )
    header [BYTE 3] := channel.id   --- any tenth from 40 up to 240
    header [BYTE 4] := dest.transp  --- destination transputer
    out := route.table [dest.transp]
    BYTE.SLICE.OUTPUT (chan[channel.id + out],header,3,1) --- ready flag
    IF
      out = 4
```

```
        SEQ
          BYTE.SLICE.OUTPUT (link4,header,1,header.size)
          BYTE.SLICE.OUTPUT (link4,message,start.byte,message.size)
      out = 5
        SEQ
          BYTE.SLICE.OUTPUT (link5,header,1,header.size)
          BYTE.SLICE.OUTPUT (link5,message,start.byte,message.size)
      out = 6
        SEQ
          BYTE.SLICE.OUTPUT (link6,header,1,header.size)
          BYTE.SLICE.OUTPUT (link6,message,start.byte,message.size)
      out = 7
        SEQ
          BYTE.SLICE.OUTPUT (link7,header,1,header.size)
          BYTE.SLICE.OUTPUT (link7,message,start.byte,message.size)
    BYTE.SLICE.OUTPUT (chan[channel.id + out],header,3,1): --- done flag


-- PROC receive (VALUE channel.id,VAR message[], message.length[])
--- ********************************************************
--- DESCRIPTION: It is an operating system routine, and it    *
--- is used to communicate between processors. It receives    *
--- the incoming message, and provides as an output parameter *
--- the size of the message just received. The parameter      *
--- channel id must have an exact match with the send         *
--- operation which originated that message.                  *
--- USAGE: receive (70,message.in,size)                       *
--- REMARK1: The user must be familiarized with the Operating *
--- System Structure before using this routine.               *
--- REMARK2: Notice that the message.length output parameter, *
--- must be a unity array of integers, while the message      *
--- itself must be declared as an array of bytes.             *
--- ********************************************************

PROC receive (VALUE channel.id,VAR message[],message.length[])=
  SEQ
    WORD.SLICE.INPUT (chan[channel.id],message.length,0,1)
    BYTE.SLICE.INPUT (chan[channel.id],message,1,message.length[0]):



-- utilities.occ
-- PROC rem.tick.to.time (VALUE start, stop, board.type, root.number)
--- ********************************************************
--- DESCRIPTION: It expects the board type which can be :    *
---   board.type = 0 ----> OPS (VAX VMS)                     *
---   board.type = 1 ----> B001 (T414:12.5 MHz)              *
---   board.type = 2 ----> B002                              *
---   board.type = 31----> B003 (T414:15 MHz - high pri)     *
---   board.type = 32----> B003 (T414:1 5MHz - low  pri)     *
---   board.type = 4 ----> B004                              *
--- and 2 signed integers representing some tick values      *
--- obtained by an assignment of the type TIME ? time.var    *
--- It then outputs the corrected elapsed time in hours, min,*
--- sec and msec, already taking into account the fact that  *
--- the timer wraps around when it reaches MAXINT or MININT. *
--- USAGE: tickk.to.time (time1,time2,31)                    *
--- REMARK: Although it takes care of the wrapping, it won't *
--- keep track of the number of times you have completed one *
--- full cycle of the timer. In order to solve this problem  *
--- you should record roughly the start time. For example, in*
--- the VAX/VMS, the full cycle of the timer is 7.2 min, so  *
--- if you get the elapsed time of 5 min 7 sec 320 msec and  *
--- you have got a rough total time of 12 minutes, then the  *
--- real total time is 12 min 19 sec 320 msec.               *
--- REMARK2: IT IS TO BE USED JUST IN REMOTE TRANSPUTERS     *
--- ********************************************************

PROC rem.tick.to.time (VALUE start, stop, board.type, root.number) =
  -- constant definitions
```

```
DEF vax.sec     =10000000 :       --- hundreds of nsec/second
DEF vax.mili    =   10000 :       --- hundreds of nsec/milisecond
DEF b001.sec    =  625000 :       --- # of 1.6 usec/second
DEF b001.mili   =     625 :       --- # of 1.6 usec/milisecond
DEF b003h.sec   = 1000000 :       --- # of usec/second
DEF b003h.mili  =    1000 :       --- # of usec/milisecond
DEF b0031.sec   =   15625 :       --- # of 64 usec/second
DEF b0031.mili  =      16 :       --- # of 64 usec/milisecond

DEF max.number.of.ticks = 2147483648 : --- maximum integer (2**31)
VAR elapsed.tick :
VAR factor1, factor2 :
VAR msec, tot.sec, sec, min, hr :

SEQ
  IF
    board.type = 0                        --- VAX VMS
      SEQ
        factor1 := vax.sec
        factor2 := vax.mili

    board.type  = 1                       --- B001
      SEQ
        factor1 := b001.sec
        factor2 := b001.mili

    board.type  = 2                       --- B002
      SKIP                                --- not implemented

    board.type  = 31                      --- B003 in high priority
      SEQ
        factor1 := b003h.sec
        factor2 := b003h.mili

    board.type  = 32                      --- B003 in low priority
      SEQ
        factor1 := b0031.sec
        factor2 := b0031.mili

    board.type  = 4                       --- B004
      SKIP                                --- not implemented

  elapsed.tick := stop - start
  IF
    elapsed.tick < 0
      elapsed.tick := elapsed.tick + max.number.of.ticks
    TRUE
      SKIP

  tot.sec := elapsed.tick/factor1
  hr      := tot.sec/3600
  min     := (tot.sec\3600)/60
  sec     := tot.sec\60
  msec    := (elapsed.tick\factor1)/factor2

  -- output time to screen
  rem.write.number (hr,root.number)
  send (40,root.number," hr ",1,0)
  rem.write.number (min,root.number)
  send (40,root.number," min ",1,0)
  rem.write.number(sec,root.number)
  send (40,root.number," sec ",1,0)
  rem.write.number(msec,root.number)
  send (40,root.number," msec",1,0):


-- PROC dump (VALUE begin.address, count, root.number)
--- ***************************************************
--- DESCRIPTION: This procedure dumps the memory starting at *
--- the given "begin.address". The value for the             *
```

```
--- "begin.address" can be either in hex or decimal.                    *
--- The count value determines how many words in memory will            *
--- be retrieved.                                                       *
--- USAGE:              a) dump (#80003540,100)                         *
---                     b) dump (1024,48)                               *
---                     c) dump (-5113,1024)                            *
--- REMARK1: When specifying the count value remember that              *
--- the retrieval is done by words, not bytes!!!                        *
--- REMARK2: If count is not a multiple of 4 it will use the            *
--- closest upper multiple.                                             *
--- REMARK3: Negatives or zero values for count although                *
--- accepted, will give you no output.                                  *
--- REMARK4: IT IS TO BE USED JUST IN REMOTE TRANSPUTERS                *
--- ********************************************************************

PROC dump (VALUE begin.address, count, root.number) =
  VAR word.read:
  VAR hex.value [10], hex.addr[10]:
  VAR address, align, times:

  SEQ
    times := 0
    rem.new.line(1,root.number)
    address := begin.address
    -- aligning a given address
    align := address\4
    IF
      align <> 0
        address := address - align
      TRUE
        SKIP
    WHILE times < count
      SEQ
        send (40,root.number,"address ",1,0)
        dec.to.hex (address,hex.addr)
        send (40,root.number,hex.addr,1,0)
        send (40,root.number,"  -->  ",1,0)
        SEQ i = [0 FOR 4]
          SEQ
            GETWORD (word.read,address)
            dec.to.hex (word.read,hex.value)
            send (40,root.number,hex.value,1,0)
            rem.space(2,root.number)
            times := times + 1
        SKIP
        address := address + 16
        rem.new.line(1,root.number)
    SKIP:


-- PROC transfer.rate (VALUE start,stop,board.type,nr.of.bytes,VAR rate)
--- ********************************************************************
--- DESCRIPTION: It is basically the same routine as                    *
--- tick.to.time, with the only difference that it returns a            *
--- rate value in Kbits/sec instead of a time value.                    *
--- USAGE: transfer.rate (time1,time2,31,4096,rate)                     *
--- REMARK: For further information refer to routine                    *
--- tick.to.time                                                        *
--- ********************************************************************

PROC transfer.rate (VALUE start,stop,board.type,nr.of.bytes,VAR rate) =
  -- constant definitions
  DEF vax.sec    =10000000 :      --- hundreds of nsec/second
  DEF b001.sec   =  625000 :      --- # of 1.6 usec/second
  DEF b003h.sec  = 1000000 :      --- # of usec/second
  DEF b003l.sec  =   15625 :      --- # of 64 usec/second
  DEF max.number.of.ticks = 2147483648 : --- maximum integer (2**31)

  -- variable declarations
  VAR elapsed.tick :
```

```
  VAR factor :        --- to convert ticks to seconds

  SEQ
    elapsed.tick := stop - start
    IF
      elapsed.tick < 0
        elapsed.tick := elapsed.tick + max.number.of.ticks
      TRUE
        SKIP
    -- selection of correct factor iaw the board
    IF
      board.type = 0                          --- VAX VMS
        factor := vax.sec

      board.type  = 1                         --- B001
        factor := b001.sec

      board.type  = 2                         --- B002
        SKIP                                  --- not implemented

      board.type  = 31                        --- B003 in high priority
        factor := b003h.sec

      board.type  = 32                        --- B003 in low priority
        factor := b003l.sec

      board.type  = 4                         --- B004
        SKIP                                  --- not implemented

    -- rate calculation
    IF
      board.type = 32
        rate := ((nr.of.bytes*8)*factor)/(elapsed.tick*1000)
        --- operation is done this way to keep precision ok!
      TRUE
        rate := ((nr.of.bytes*8)*(factor/1000))/elapsed.tick
        --- operation is done this way in order to not exceed maxint
        --- on the numerator.
        --- multiply by 8 due to 8 bits per byte
        --- divide by 1000 to have the tranfer.rate in kbits/sec

    SKIP:


-- PROC operating.system
PROC operating.system =
  -- PROC input.handler
  PROC input.handler =
    -- variable and constants declarations
    VAR header0 [BYTE 5],
        header1 [BYTE 5],
        header2 [BYTE 5],
        header3 [BYTE 5],

        buffer.in0 [BYTE max.block.size],
        buffer.in1 [BYTE max.block.size],
        buffer.in2 [BYTE max.block.size],
        buffer.in3 [BYTE max.block.size],

        block.size0[1], out0,
        block.size1[1], out1,
        block.size2[1], out2,
        block.size3[1], out3:

    SEQ
      -- initializing the buffers
      SEQ i = [0 FOR max.block.size]
        SEQ
```

```
    buffer.in0 [BYTE i] := '0'
    buffer.in1 [BYTE i] := '1'
    buffer.in2 [BYTE i] := '2'
    buffer.in3 [BYTE i] := '3'
SKIP

PAR
  WHILE TRUE
    -- listen to link0
    SEQ
      -- receiving the header
      BYTE.SLICE.INPUT (link0,header0,1,header.size)

      -- decoding the block size
      block.size0[0] := ((256*header0[BYTE 1])+header0[BYTE 2])

      -- buffering the message
      BYTE.SLICE.INPUT (link0,buffer.in0,1,block.size0[0])

      IF
        -- the message is to be bypassed
        header0 [BYTE 4] <> this.transputer
          SEQ
            -- finding the best link to output that message
            out0 := route.table [header0 [BYTE 4]]

            -- outputing to the required link
            --- request flag thru chan 4, 5, 6 or 7
            BYTE.SLICE.OUTPUT(chan[out0],header0,3,1)
            IF
              out0 = 4
                SEQ
                  BYTE.SLICE.OUTPUT (link4,header0,1,header.size)
                  BYTE.SLICE.OUTPUT (link4,buffer.in0,1,
                                                 block.size0[0])
              out0 = 5
                SEQ
                  BYTE.SLICE.OUTPUT (link5,header0,1,header.size)
                  BYTE.SLICE.OUTPUT (link5,buffer.in0,1,
                                                 block.size0[0])
              out0 = 6
                SEQ
                  BYTE.SLICE.OUTPUT (link6,header0,1,header.size)
                  BYTE.SLICE.OUTPUT (link6,buffer.in0,1,
                                                 block.size0[0])
              out0 = 7
                SEQ
                  BYTE.SLICE.OUTPUT (link7,header0,1,header.size)
                  BYTE.SLICE.OUTPUT (link7,buffer.in0,1,
                                                 block.size0[0])
            --- release flag
            BYTE.SLICE.OUTPUT(chan[out0],header0,3,1)

        -- the message is for this transputer
        header0 [BYTE 4] = this.transputer
          SEQ
            -- passing the size of the message (block.size0[0])
            WORD.SLICE.OUTPUT (chan[header0[BYTE 3]],
                                                 block.size0,0,1)

            -- passing the message itself
            BYTE.SLICE.OUTPUT (chan[header0[BYTE 3]],buffer.in0,1,
                                                 block.size0[0])

  WHILE TRUE
    -- listen to link1
    SEQ
      -- receiving the header
      BYTE.SLICE.INPUT (link1,header1,1,header.size)
```

```
        -- decoding the block size
        block.size1[0] := ((256 * header1[BYTE 1])+header1[BYTE 2])

        -- buffering the message
        BYTE.SLICE.INPUT (link1,buffer.in1,1,block.size1[0])

        IF
          -- the message is to be bypassed
          header1 [BYTE 4] <> this.transputer
            SEQ
              -- finding the best link to output that message
              out1 := route.table [header1 [BYTE 4]]

              -- outputing to the required link
              --- request flag thru chan 14, 15, 16 or 17
              BYTE.SLICE.OUTPUT(chan[10+out1],header1,3,1)
              IF
                out1 = 4
                  SEQ
                    BYTE.SLICE.OUTPUT (link4,header1,1,header.size)
                    BYTE.SLICE.OUTPUT (link4,buffer.in1,1,
                                                block.size1[0])
                out1 = 5
                  SEQ
                    BYTE.SLICE.OUTPUT (link5,header1,1,header.size)
                    BYTE.SLICE.OUTPUT (link5,buffer.in1,1,
                                                block.size1[0])
                out1 = 6
                  SEQ
                    BYTE.SLICE.OUTPUT (link6,header1,1,header.size)
                    BYTE.SLICE.OUTPUT (link6,buffer.in1,1,
                                                block.size1[0])
                out1 = 7
                  SEQ
                    BYTE.SLICE.OUTPUT (link7,header1,1,header.size)
                    BYTE.SLICE.OUTPUT (link7,buffer.in1,1,
                                                block.size1[0])
              --- release flag
              BYTE.SLICE.OUTPUT(chan[10+out1],header1,3,1)

          -- the message is for this transputer
          header1 [BYTE 4] = this.transputer
            SEQ
              -- passing the size of the message (block.size1[0])
              WORD.SLICE.OUTPUT (chan[header1 [BYTE 3]],
                                                block.size1,0,1)

              -- passing the message itself
              BYTE.SLICE.OUTPUT (chan[header1 [BYTE 3]],buffer.in1,
                                                1,block.size1[0])

      WHILE TRUE
        -- listen to link2
        SEQ
          -- receiving the header
          BYTE.SLICE.INPUT (link2,header2,1,header.size)

          -- decoding the block size
          block.size2[0] := ((256 * header2[BYTE 1])+header2[BYTE 2])

          -- buffering the message
          BYTE.SLICE.INPUT (link2,buffer.in2,1,block.size2[0])

          IF
            -- the message is to be bypassed
            header2 [BYTE 4] <> this.transputer
              SEQ
                -- finding the best link to output that message
                out2 := route.table [header2 [BYTE 4]]
```

```
            -- outputing to the required link
            --- request flag thru chan 24, 25, 26 or 27
            BYTE.SLICE.OUTPUT(chan[20+out2],header2,3,1)
            IF
              out2 = 4
                SEQ
                  BYTE.SLICE.OUTPUT (link4,header2,1,header.size)
                  BYTE.SLICE.OUTPUT (link4,buffer.in2,1,
                                                 block.size2[0])
              out2 = 5
                SEQ
                  BYTE.SLICE.OUTPUT (link5,header2,1,header.size)
                  BYTE.SLICE.OUTPUT (link5,buffer.in2,1,
                                                 block.size2[0])
              out2 = 6
                SEQ
                  BYTE.SLICE.OUTPUT (link6,header2,1,header.size)
                  BYTE.SLICE.OUTPUT (link6,buffer.in2,1,
                                                 block.size2[0])
              out2 = 7
                SEQ
                  BYTE.SLICE.OUTPUT (link7,header2,1,header.size)
                  BYTE.SLICE.OUTPUT (link7,buffer.in2,1,
                                                 block.size2[0])
            --- release flag
            BYTE.SLICE.OUTPUT(chan[20+out2],header2,3,1)

        -- the message is for this transputer
        header2 [BYTE 4] = this.transputer
          SEQ
            -- passing the size of the message (block.size2[0])
            WORD.SLICE.OUTPUT (chan[header2 [BYTE 3]],
                                                 block.size2,0,1)

            -- passing the message itself
            BYTE.SLICE.OUTPUT (chan[header2 [BYTE 3]],buffer.in2,
                                                 1,block.size2[0])

WHILE TRUE
  -- listen to link3
  SEQ
    -- receiving the header
    BYTE.SLICE.INPUT (link3,header3,1,header.size)

    -- decoding the block size
    block.size3[0] := ((256 * header3[BYTE 1])+header3[BYTE 2])

    -- buffering the message
    BYTE.SLICE.INPUT (link3,buffer.in3,1,block.size3[0])

    IF
      -- the message is to be bypassed
      header3 [BYTE 4] <> this.transputer
        SEQ
          -- finding the best link to output that message
          out3 := route.table [header3 [BYTE 4]]

          -- outputing to the required link
          --- request flag thru chan 34, 35, 36 or 37
          BYTE.SLICE.OUTPUT(chan[30+out3],header3,3,1)
          IF
            out3 = 4
              SEQ
                BYTE.SLICE.OUTPUT (link4,header3,1,header.size)
                BYTE.SLICE.OUTPUT (link4,buffer.in3,1,
                                                 block.size3[0])
            out3 = 5
              SEQ
                BYTE.SLICE.OUTPUT (link5,header3,1,header.size)
                BYTE.SLICE.OUTPUT (link5,buffer.in3,1,
```

```
                                                block.size3[0])
                        out3 = 6
                          SEQ
                            BYTE.SLICE.OUTPUT (link6,header3,1,header.size)
                            BYTE.SLICE.OUTPUT (link6,buffer.in3,1,
                                                block.size3[0])
                        out3 = 7
                          SEQ
                            BYTE.SLICE.OUTPUT (link7,header3,1,header.size)
                            BYTE.SLICE.OUTPUT (link7,buffer.in3,1,
                                                block.size3[0])
                      --- release flag
                      BYTE.SLICE.OUTPUT(chan[30+out3],header3,3,1)

                -- the message is for this transputer
                header3 [BYTE 4] = this.transputer
                  SEQ
                    -- passing the size of the message (block.size3[0])
                    WORD.SLICE.OUTPUT (chan[header3 [BYTE 3]],
                                                block.size3,0,1)

                    -- passing the message itself
                    BYTE.SLICE.OUTPUT (chan[header3 [BYTE 3]],buffer.in3,
                                                1,block.size3[0]) :


-- PROC output.handler
PROC output.handler =
  -- local variable declarations
  VAR flag4 [BYTE 2]:
  VAR flag5 [BYTE 2]:
  VAR flag6 [BYTE 2]:
  VAR flag7 [BYTE 2]:

  PAR
    WHILE TRUE
      ALT i = [0 FOR max.io.channels]
        chan [(10*i) +4] ? flag4 [BYTE 0] --- for link4
          BYTE.SLICE.INPUT (chan [(10*i) +4],flag4,0,1)
    WHILE TRUE
      ALT j = [0 FOR max.io.channels]
        chan [(10*j) +5] ? flag5 [BYTE 0] --- for link5
          BYTE.SLICE.INPUT (chan [(10*j) +5],flag5,0,1)
    WHILE TRUE
      ALT k = [0 FOR max.io.channels]
        chan [(10*k) +6] ? flag6 [BYTE 0] --- for link6
          BYTE.SLICE.INPUT (chan [(10*k) +6],flag6,0,1)
    WHILE TRUE
      ALT l = [0 FOR max.io.channels]
        chan [(10*l) +7] ? flag7 [BYTE 0] --- for link7
          BYTE.SLICE.INPUT (chan [(10*l) +7],flag7,0,1):
```

```
-- main body of the operating system
SEQ
  -- receiving the routing table
  route.table[0]   := t0          --- output link to transp #0
  route.table[1]   := t1          --- output link to transp #1
  route.table[2]   := t2          --- output link to transp #2
  route.table[3]   := t3          --- output link to transp #3
  route.table[4]   := t4          --- output link to transp #4
  route.table[5]   := t5          --- output link to transp #5
  route.table[6]   := t6          --- output link to transp #6
  route.table[7]   := t7          --- output link to transp #7
  route.table[8]   := t8          --- output link to transp #8
  route.table[9]   := t9          --- output link to transp #9
  route.table[10]  := t10         --- output link to transp #10
  route.table[11]  := t11         --- output link to transp #11
  route.table[12]  := t12         --- output link to transp #12
  route.table[13]  := t13         --- output link to transp #13
  route.table[14]  := t14         --- output link to transp #14
  route.table[15]  := t15         --- output link to transp #15
  route.table[16]  := t16         --- output link to transp #16
  route.table[17]  := t17         --- output link to transp #17

  PAR
    output.handler
    input.handler:
```

APPENDIX F

THE EVALUATION PROGRAM FOR THE OPERATING SYSTEM (EVAL_OS.TDS)

```
-- PROGRAM os.evaluation
-- os.evaluation
-- SC PROC hostproc
-- PROC hostproc(CHAN A,B,C,D,E,F,G,H,VALUE this.transputer,route.table)
PROC hostproc(CHAN A,B,C,D,E,F,G,H,
              VALUE this.transputer,
                    t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,
                    t10,t11,t12,t13,t14,t15,t16,t17) =

  -- root_os.tds
  --- ********************************************************
  --- In this place should be imbedded the filed fold        *
  --- ROOT_OS.TDS, which contains the source code of the     *
  --- operating system for the root transputer.              *
  --- It is fully documented in Appendix 5.                  *
  --- ********************************************************


  -- PROC user.interface
  PROC user.interface =
    -- constant and variable declarations
    DEF sizetable = TABLE [ 1, 2, 4, 8, 16, 32, 64, 128, 256,
                            512, 1024, 1280, 2048, 4096 ]:
    DEF nr.of.sizes = 14:       --- # of entries in the above table
    DEF maxblock.size = 4096:   --- max from the above table
    VAR buffer0 [BYTE maxblock.size + 1],
        buffer1 [BYTE maxblock.size + 1],
        buffer2 [BYTE maxblock.size + 1],
        buffer3 [BYTE maxblock.size + 1]:
    VAR  run                :   ---  number of runs made (RUN #)
    VAR  answer [BYTE 2] :      ---  user's choice in continue or quit
    VAR  repetition :           ---  number of times to carry each xfer
    VAR  dummy0[1],dummy1[1],dummy2[1],dummy3[1] :

    -- PROC write.header
    PROC write.header =
    --- writes the header of the output table
      SEQ
        run := run + 1
        clear.screen
        write.string ("RUN # ")
        write.number (run)
        space(3)
        write.string ("CPUs  IDLING                ")
        space(2)
        write.string ("BYTE.SLICE.input/output")
        space(2)
        write.string ("Repetition = ")
        write.number (repetition)
        new.line(2)
        write.string ("BYTES  1OUT 1IN 2OUT 2IN 3OUT 3IN ")
        write.string ("4OUT 4IN 4INOUT")
        new.line(1):


    -- PROC transfer
    PROC transfer =
      -- variable declarations
      VAR block.size,
          actual.rate,
```

```
    rate,
    ch0[BYTE 2],ch1[BYTE 2],ch2[BYTE 2],ch3[BYTE 2],
    time0[4],
    time1[4]:

SEQ
  SEQ i = [0 FOR nr.of.sizes]
    SEQ
      -- making the table after each io operation
      block.size := sizetable[i]
      write.number (block.size)
      tab (1)
      -- output to one channel
      actual.rate := 0
      SEQ j = [1 FOR repetition]
        SEQ
          send (90,0,"a ",1,1)
          TIME ? time0[0]
          send (90,0,buffer0,1,block.size)
          TIME ? time1[0]
          transfer.rate(time0[0],time1[0],1,block.size,rate)
          actual.rate := ((actual.rate * (j-1)) + rate)/j
      SKIP
      write.number (actual.rate)
      tab (1)

      -- input from one channel
      actual.rate := 0
      SEQ j = [1 FOR repetition]
        SEQ
          send (90,0,"a ",1,1)
          TIME ? time0[0]
          receive(50,buffer0,dummy0)
          TIME ? time1[0]
          transfer.rate(time0[0],time1[0],1,block.size,rate)
          actual.rate := ((actual.rate * (j-1)) + rate)/j
      SKIP
      write.number (actual.rate)
      tab (1)

      -- output to two channels
      actual.rate := 0
      SEQ j = [1 FOR repetition]
        SEQ
          PAR
            send (90,0,"a ",1,1)
            send (100,1,"a ",1,1)
          TIME ? time0[0]
          PAR
            send(90,0,buffer0,1,block.size)
            send(100,1,buffer1,1,block.size)
          TIME ? time1[0]
          transfer.rate(time0[0],time1[0],1,block.size,rate)
          actual.rate := ((actual.rate * (j-1)) + rate)/j
      SKIP
      write.number (actual.rate)
      tab (1)

      -- input from two channels
      actual.rate := 0
      SEQ j = [1 FOR repetition]
        SEQ
          PAR
            send (90,0,"a ",1,1)
            send (100,1,"a ",1,1)
          TIME ? time0[0]
          PAR
            receive(50,buffer0,dummy0)
            receive(60,buffer1,dummy1)
          TIME ? time1[0]
```

```
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
tab (1)

-- output to three channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      send (90,0,"a ",1,1)
      send (100,1,"a ",1,1)
      send (110,2,"a ",1,1)
    TIME ? time0[0]
    PAR
      send(90,0,buffer0,1,block.size)
      send(100,1,buffer1,1,block.size)
      send(110,2,buffer2,1,block.size)
    TIME ? time1[0]
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
tab (1)

-- input from three channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      send (90,0,"a ",1,1)
      send (100,1,"a ",1,1)
      send (110,2,"a ",1,1)
    TIME ? time0[0]
    PAR
      receive(50,buffer0,dummy0)
      receive(60,buffer1,dummy1)
      receive(70,buffer2,dummy2)
    TIME ? time1[0]
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
tab (1)

-- output to four channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      send (90,0,"a ",1,1)
      send (100,1,"a ",1,1)
      send (110,2,"a ",1,1)
      send (120,3,"a ",1,1)
    TIME ? time0[0]
    PAR
      send(90,0,buffer0,1,block.size)
      send(100,1,buffer1,1,block.size)
      send(110,2,buffer2,1,block.size)
      send(120,3,buffer3,1,block.size)
    TIME ? time1[0]
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
tab (1)

-- input from four channels
actual.rate := 0
```

```
        SEQ j = [1 FOR repetition]
          SEQ
            PAR
              send (90,0,"a ",1,1)
              send (100,1,"a ",1,1)
              send (110,2,"a ",1,1)
              send (120,3,"a ",1,1)
            TIME ? time0[0]
            PAR
              receive(50,buffer0,dummy0)
              receive(60,buffer1,dummy1)
              receive(70,buffer2,dummy2)
              receive(80,buffer3,dummy3)
            TIME ? time1[0]
            transfer.rate(time0[0],time1[0],1,block.size,rate)
            actual.rate := ((actual.rate * (j-1)) + rate)/j
        SKIP
        write.number (actual.rate)
        tab (1)

        -- all output and input in parallel
        actual.rate := 0
        SEQ j = [1 FOR repetition]
          SEQ
            PAR
              send (90,0,"a ",1,1)
              send (100,1,"a ",1,1)
              send (110,2,"a ",1,1)
              send (120,3,"a ",1,1)
            TIME ? time0[0]
            PAR
              send(90,0,buffer0,1,block.size)
              send(100,1,buffer1,1,block.size)
              send(110,2,buffer2,1,block.size)
              send(120,3,buffer3,1,block.size)
              receive(50,buffer0,dummy0)
              receive(60,buffer1,dummy1)
              receive(70,buffer2,dummy2)
              receive(80,buffer3,dummy3)
            TIME ? time1[0]
            transfer.rate(time0[0],time1[0],1,block.size,rate)
            actual.rate := ((actual.rate * (j-1)) + rate)/j
        SKIP
        write.number (actual.rate)
        new.line(1)
    SKIP
    new.line(1):


-- main program
SEQ
  -- some variables initializations
  run := 0
  answer [BYTE 1] := 'z'
  repetition := 20

  -- initialization of buffers with bytes
  SEQ k = [1 FOR maxblock.size + 1]
    SEQ
      buffer0 [BYTE k] := '0'
      buffer1 [BYTE k] := '1'
      buffer2 [BYTE k] := '2'
      buffer3 [BYTE k] := '3'
  SKIP

  -- program explanation
  clear.screen
  write.string (" This is an Evaluation Program for the Transputer")      ")
  new.line(2)
  write.string (" The table presents transfer rates in Kbits/sec")
```

```
      new.line(1)
      write.string (" for 14 block.sizes in 9 channel combinations ")
      new.line(2)
      write.string (" TYPE (Y)ES  if you want to use it ")
      new.line(1)
      write.string ("      (N)O   if you want to quit ")
      new.line(1)

      -- validate answer
      WHILE ((answer [BYTE 1] <> 'Y') AND (answer [BYTE 1] <> 'N'))
        SEQ
          write.string (" Type your choice ")
          Keyboard ? answer [BYTE 1]
          capitalize (answer)
          screen[4] ! 'a'
          Screen ! answer [BYTE 1]
          screen[4] ! 'a'
          new.line(1)

      WHILE (answer [BYTE 1] = 'Y')
        SEQ
          -- writing the table header
          write.header

          -- running the actual transfer program
          transfer

          -- prompting for new run
          answer [BYTE 1] := 'Z'  --- to make the next loop be executed
          -- another run
          WHILE ((answer [BYTE 1] <> 'Y') AND (answer [BYTE 1] <> 'N'))
            SEQ
              write.string (" Do you want another run ? (Y)ES or (N)O ")
              Keyboard ? answer [BYTE 1]
              capitalize (answer)
              screen[4] ! 'a'
              Screen ! answer [BYTE 1]
              screen[4] ! 'a'
              new.line(1)

          PAR
            send (90,0,answer,1,1)
            send (100,1,answer,1,1)
            send (110,2,answer,1,1)
            send (120,3,answer,1,1)

      -- exiting the program
      clear.screen
      write.string (" Press reset button to get back to VAX/VMS ") :


PAR
  operating.system
  user.interface:
```

```
-- SC PROC transfer0.b003
-- PROC transfer0.b003 (CHAN A,B,C,D,VALUE this.transputer,route.table)
PROC transfer0.b003 (CHAN A,B,C,D,
                     VALUE this.transputer,
                           t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,
                           t10,t11,t12,t13,t14,t15,t16,t17) =

  -- remote_os.tds
  --- ********************************************************
  --- In this place should be imbedded the filed fold        *
  --- REMOTE_OS.TDS, which contains the source code of the   *
  --- operating system for remote transputers.               *
  --- It is fully documented in Appendix E.                  *
  --- ********************************************************


  -- PROC user0
  PROC user0 =
    -- constants and variables declarations
    DEF sizetable = TABLE [ 1, 2, 4, 8, 16, 32, 64, 128, 256, 512,
                            1024, 1280, 2048, 4096 ]:
    DEF nr.of.sizes = 14:          --- # of entries in the above table
    DEF maxblock.size = 4096:      --- max from the above table
    VAR  answer [BYTE 2] :         ---  user's choice in continue or quit
    VAR  dummy0[1] :
    VAR  repetition :

    -- PROC transfer0
    PROC transfer0 =
      -- variable declarations
      VAR block.size,
          ch0 [BYTE 2]:
      VAR buffer0 [BYTE maxblock.size + 1]:

      SEQ
        -- initialization of buffers
        SEQ k = [1 FOR maxblock.size + 1]
          SEQ
            buffer0 [BYTE k] := '0'
        SKIP

        SEQ i = [0 FOR nr.of.sizes]
          SEQ
            block.size := sizetable[i]
            -- input and output handling
            -- input from one channel
            SEQ j = [1 FOR repetition]
              SEQ
                receive (90,ch0,dummy0)
                receive (90,buffer0,dummy0)
            SKIP

            -- output to one channel
            SEQ j = [1 FOR repetition]
              SEQ
                receive (90,ch0,dummy0)
                send(50,10,buffer0,1,block.size)
            SKIP

            -- input from two channels
            SEQ j = [1 FOR repetition]
              SEQ
                receive (90,ch0,dummy0)
                receive (90,buffer0,dummy0)
            SKIP

            -- output to two channels
            SEQ j = [1 FOR repetition]
              SEQ
                receive (90,ch0,dummy0)
```

```
            send(50,10,buffer0,1,block.size)
          SKIP

          -- input from three channels
          SEQ j = [1 FOR repetition]
            SEQ
              receive (90,ch0,dummy0)
              receive (90,buffer0,dummy0)
          SKIP

          -- output to three channels
          SEQ j = [1 FOR repetition]
            SEQ
              receive (90,ch0,dummy0)
              send(50,10,buffer0,1,block.size)
          SKIP

          -- input from four channels
          SEQ j = [1 FOR repetition]
            SEQ
              receive (90,ch0,dummy0)
              receive (90,buffer0,dummy0)
          SKIP

          -- output to four channels
          SEQ j = [1 FOR repetition]
            SEQ
              receive (90,ch0,dummy0)
              send(50,10,buffer0,1,block.size)
          SKIP

          -- all output and input in parallel
          SEQ j = [1 FOR repetition]
            SEQ
              receive (90,ch0,dummy0)
              PAR
                receive (90,buffer0,dummy0)
                send(50,10,buffer0,1,block.size)
          SKIP
      SKIP:

  -- main program
  SEQ
    repetition := 20
    answer[BYTE 1] := 'Y'
    WHILE  answer[BYTE 1] = 'Y'
      SEQ
        transfer0
        receive (90,answer,dummy0) :


PAR
  operating.system
  user0 :
```

```
-- SC PROC transfer1.b003
-- PROC transfer1.b003 (CHAN A,B,C,D, VALUE this.transputer, route.table)
PROC transfer1.b003 (CHAN A,B,C,D,
                     VALUE this.transputer,
                           t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,
                           t10,t11,t12,t13,t14,t15,t16,t17) =

  -- remote_os.tds
  --- ********************************************************
  --- In this place should be imbedded the filed fold        *
  --- REMOTE_OS.IDS, which contains the source code of the   *
  --- operating system for remote transputers.               *
  --- It is fully documented in Appendix E.                  *
  --- ********************************************************


  -- PROC user1
  PROC user1 =
    -- constants and variables declarations
    DEF sizetable = TABLE [ 1, 2, 4, 8, 16, 32, 64, 128, 256, 512,
                            1024, 1280, 2048, 4096 ]:
    DEF nr.of.sizes = 14:          --- # of entries in the above table
    DEF maxblock.size = 4096:      --- max from the above table
    VAR  answer [BYTE 2] :         ---  user's choice in continue or quit
    VAR  dummy0[1] :
    VAR  repetition :

    -- PROC transfer0
    PROC transfer0 =
      -- variable declarations
      VAR block.size,
          ch0 [BYTE 2]:
      VAR buffer0 [BYTE maxblock.size + 1]:


      SEQ
        -- initialization of buffers
        SEQ k = [1 FOR maxblock.size + 1]
          SEQ
            buffer0 [BYTE k] := '0'
        SKIP

        SEQ i = [0 FOR nr.of.sizes]
          SEQ
            block.size := sizetable[i]
            -- input and output handling
            -- input from two channels
            SEQ j = [1 FOR repetition]
              SEQ
                receive (100,ch0,dummy0)
                receive (100,buffer0,dummy0)
            SKIP

            -- output to two channels
            SEQ j = [1 FOR repetition]
              SEQ
                receive (100,ch0,dummy0)
                send(60,10,buffer0,1,block.size)
            SKIP

            -- input from three channels
            SEQ j = [1 FOR repetition]
              SEQ
                receive (100,ch0,dummy0)
                receive (100,buffer0,dummy0)
            SKIP

            -- output to three channels
            SEQ j = [1 FOR repetition]
              SEQ
```

```
            receive (100,ch0,dummy0)
            send(60,10,buffer0,1,block.size)
        SKIP

        -- input from four channels
        SEQ j = [1 FOR repetition]
          SEQ
            receive (100,ch0,dummy0)
            receive (100,buffer0,dummy0)
        SKIP

        -- output to four channels
        SEQ j = [1 FOR repetition]
          SEQ
            receive (100,ch0,dummy0)
            send(60,10,buffer0 1,block.size)
        SKIP

        -- all output and input in parallel
        SEQ j = [1 FOR repetition]
          SEQ
            receive (100,ch0,dummy0)
            PAR
              receive (100,buffer0,dummy0)
              send(60,10,buffer0,1,block.size)
        SKIP
    SKIP:

  -- main program
  SEQ
    repetition := 20
    answer[BYTE 1] := 'Y'
    WHILE  answer[BYTE 1] = 'Y'
      SEQ
        transfer0
        receive (100,answer,dummy0) :


PAR
  operating.system
  user1 :
```

```
-- SC PROC transfer2.b003
-- PROC transfer2.b003 (CHAN A,B,C,D,VALUE this.transputer,route.table)
PROC transfer2.b003 (CHAN A,B,C,D,
                     VALUE this.transputer,
                           t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,
                           t10,t11,t12,t13,t14,t15,t16,t17) =

  -- remote_os.tds
  --- ********************************************************
  --- In this place should be imbedded the filed fold        *
  --- REMOTE_OS.TDS, which contains the source code of the   *
  --- operating system for remote transputers.               *
  --- It is fully documented in Appendix E.                  *
  --- ********************************************************


  -- PROC user2
  PROC user2 =
    -- constants and variables declarations
    DEF sizetable = TABLE [ 1, 2, 4, 8, 16, 32, 64, 128, 256, 512,
                            1024, 1280, 2048, 4096 ]:
    DEF nr.of.sizes = 14:       --- # of entries in the above table
    DEF maxblock.size = 4096:   --- max from the above table
    VAR  answer [BYTE 2] :      ---  user's choice in continue or quit
    VAR  dummy0[1] :
    VAR  repetition :

    -- PROC transfer0
    PROC transfer0 =
      -- variable declarations
      VAR block.size,
          ch0 [BYTE 2]:
      VAR buffer0 [BYTE maxblock.size + 1]:


      SEQ
        -- initialization of buffers
        SEQ k = [1 FOR maxblock.size + 1]
          SEQ
            buffer0 [BYTE k] := '0'
        SKIP

        SEQ i = [0 FOR nr.of.sizes]
          SEQ
            block.size := sizetable[i]
            -- input and output handling
            -- input from three channels
            SEQ j = [1 FOR repetition]
              SEQ
                receive (110,ch0,dummy0)
                receive (110,buffer0,dummy0)
            SKIP

            -- output to three channels
            SEQ j = [1 FOR repetition]
              SEQ
                receive (110,ch0,dummy0)
                send(70,10,buffer0,1,block.size)
            SKIP

            -- input from four channels
            SEQ j = [1 FOR repetition]
              SEQ
                receive (110,ch0,dummy0)
                receive (110,buffer0,dummy0)
            SKIP

            -- output to four channels
            SEQ j = [1 FOR repetition]
              SEQ
```

```
            receive (110,ch0,dummy0)
            send(70,10,buffer0,1,block.size)
        SKIP

        -- all output and input in parallel
        SEQ j = [1 FOR repetition]
          SEQ
            receive (110,ch0,dummy0)
            PAR
              receive (110,buffer0,dummy0)
              send(70,10,buffer0,1,block.size)
        SKIP
    SKIP:

  -- main program
  SEQ
    repetition := 20
    answer[BYTE 1] := 'Y'
    WHILE answer[BYTE 1] = 'Y'
      SEQ
        transfer0
        receive (110,answer,dummy0) :


PAR
  operating.system
  user2:
```

```
-- SC PROC transfer3.b003
-- PROC transfer3.b003 (CHAN A,B,C,D,VALUE this.transputer,route.table)
PROC transfer3.b003 (CHAN A,B,C,D,
                     VALUE this.transputer,
                           t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,
                           t10,t11,t12,t13,t14,t15,t16,t17) =

  -- remote_os.tds
  --- ********************************************************
  --- In this place should be imbedded the filed fold        *
  --- REMOTE_OS.TDS, which contains the source code of the   *
  --- operating system for remote transputers.               *
  --- It is fully documented in Appendix E.                  *
  --- ********************************************************


  -- PROC user3
  PROC user3 =
    -- constants and variables declarations
    DEF sizetable = TABLE [ 1, 2, 4, 8, 16, 32, 64, 128, 256, 512,
                            1024, 1280, 2048, 4096 ]:
    DEF nr.of.sizes = 14:       --- # of entries in the above table
    DEF maxblock.size = 4096:   --- max from the above table
    VAR  answer [BYTE 2] :      ---  user's choice in continue or quit
    VAR  dummy0[1] :
    VAR  repetition :

    -- PROC transfer0
    PROC transfer0 =
      -- variable declarations
      VAR block.size,
          ch0 [BYTE 2]:
      VAR buffer0 [BYTE maxblock.size + 1]:


      SEQ
        -- initialization of buffers
        SEQ k = [1 FOR maxblock.size + 1]
          SEQ
            buffer0 [BYTE k] := '0'
        SKIP

        SEQ i = [0 FOR nr.of.sizes]
          SEQ
            block.size := sizetable[i]
            -- input and output handling
            -- input from four channels
            SEQ j = [1 FOR repetition]
              SEQ
                receive (120,ch0,dummy0)
                receive (120,buffer0,dummy0)
            SKIP

            -- output to four channels
            SEQ j = [1 FOR repetition]
              SEQ
                receive (120,ch0,dummy0)
                send(80,10,buffer0,1,block.size)
            SKIP

            -- all output and input in parallel
            SEQ j = [1 FOR repetition]
              SEQ
                receive (120,ch0,dummy0)
                PAR
                  receive (120,buffer0,dummy0)
                  send(80,10,buffer0,1,block.size)
            SKIP
        SKIP:
```

```
  -- main program
  SEQ
    repetition := 20
    answer[BYTE 1] := 'Y'
    WHILE  answer[BYTE 1] = 'Y'
      SEQ
        transfer0
        receive (120,answer,dummy0) :

PAR
  operating.system
  user3 :
```

```
-- configuration
-- Link Definitions
DEF link0out = 0 :
DEF link1out = 1 :
DEF link2out = 2 :
DEF link3out = 3 :
DEF link0in  = 4 :
DEF link1in  = 5 :
DEF link2in  = 6 :
DEF link3in  = 7 :

-- Variables and Constants Declarations
DEF  root = 10:
DEF  max.pipes = 20:
CHAN pipe [max.pipes]:


PLACED PAR
  -- PROCESSOR root
  PROCESSOR root
    PLACE pipe[0]   AT link0in  :
    PLACE pipe[2]   AT link1in  :
    PLACE pipe[4]   AT link2in  :
    PLACE pipe[6]   AT link3in  :
    PLACE pipe[1]   AT link0out :
    PLACE pipe[3]   AT link1out :
    PLACE pipe[5]   AT link2out :
    PLACE pipe[7]   AT link3out :

    hostproc (pipe[0],pipe[2],pipe[4],pipe[6],
              pipe[1],pipe[3],pipe[5],pipe[7],
              10,
              4,5,6,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

  -- PROCESSOR 0
  PROCESSOR 0
    PLACE pipe[1]   AT link0in  :
    PLACE pipe[9]   AT link1in  :
    PLACE pipe[0]   AT link0out :
    PLACE pipe[8]   AT link1out :

    transfer0.b003 (pipe[1],pipe[9],pipe[0],pipe[8],
                    0,
                    0,6,5,7,0,0,0,0,0,0,4,0,0,0,0,0,0,0)

  -- PROCESSOR 1
  PROCESSOR 1
    PLACE pipe[3]   AT link0in  :
    PLACE pipe[10]  AT link1in  :
    PLACE pipe[2]   AT link0out :
    PLACE pipe[11]  AT link1out :

    transfer1.b003 (pipe[3],pipe[10],pipe[2],pipe[11],
                    1,
                    7,0,6,5,0,0,0,0,0,0,4,0,0,0,0,0,0,0)

  -- PROCESSOR 2
  PROCESSOR 2
    PLACE pipe[5]   AT link0in  :
    PLACE pipe[8]   AT link1in  :
    PLACE pipe[4]   AT link0out :
    PLACE pipe[9]   AT link1out :

    transfer2.b003 (pipe[5],pipe[8],pipe[4],pipe[9],
                    2,
                    5,7,0,6,0,0,0,0,0,0,4,0,0,0,0,0,0,0)
```

```
-- PROCESSOR 3
PROCESSOR 3
  PLACE pipe[7]   AT link0in  :
  PLACE pipe[11]  AT link1in  :
  PLACE pipe[6]   AT link0out :
  PLACE pipe[10]  AT link1out :

  transfer3.b003 (pipe[7],pipe[11],pipe[6],pipe[10],
                  3,
                  6,5,7,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0)
```

LIST OF REFERENCES

1. Garret, D. R., *A Software System Implementation Guide and System Prototyping Facility for the MCORTEX Executive on the Real Time Cluster*, M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1986.

2. Rowe, W. R., *Adaption of MCORTEX to the AEGIS Simulation Environment*, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1984.

3. INMOS Limited, *The Transputer Family*, June 1986.

4. INMOS Limited, *IMS B001 Evaluation Board User Manual*, 1985.

5. INMOS Limited, *IMS B003 Evaluation Board User Manual*, 1985.

6. INMOS Limited, *IMS B004 Evaluation Board User Manual*, 1985.

7. Evin, B. , *Implementation of a Serial Delay Insertion Type Loop Communication for a Real Time Multitransputer System*, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

8. Selcuk, Z., *Implementation of a Serial Communication Process for a Fault Tolerant, Real Time, Multitransputer Operating System*, M.S. Thesis, Naval Postgraduate School, Monterey, California, 1984.

9. Vanni, J. F., *Test and Evaluation of the Transputer in a Multitransputer System Configuration*, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1987.

10. INMOS Limited, *IMS D600 Transputer Development System*, 1985.

11. Shepherd Roger, *Extraordinary use of transputer links*, INMOS Technical note 1, November 1986.

BIBLIOGRAPHY

Heath M. T., *The Hypercube: A Tutorial Overview*, Oak Ridge National Laboratory, 1986.

INMOS Corporation, *Compiler Writers Guide*, Draft, 1986.

INMOS Limited, *OCCAM Programming Manual*, 1983.

INMOS Limited, *Transputer Reference Manual*, October 1986.

Peterson & Silberschatz, *Operating Systems Concepts*, Addison-Wesley Publishing Co., Inc., 1983.

Tanenbaum A. S., *Computer Networks*, Prentice Hall, New Jersey, 1981.

Weitzman C., *Distributed Micro/Minicomputer Systems*, Prentice-Hall, New Jersey, 1980.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
4.	Dr. Uno R. Kodres, Code 52Kr Department of Computer Science Naval Postgraduate School Monterey, CA 93943	3
5.	Dr. Daniel L. Davis, Code 52Dv Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
6.	Daniel Green, Code 20F Naval Surface Weapons Center Dahlgren, VA 22449	1
7.	Jerry Gaston, Code N24 Naval Surface Weapons Center Dahlgren, VA 22449	1
8.	CAPT. J. Hood, USN PMS 400B5 Naval Sea Systems Command Washington D.C. 20362	1
9.	RCA AEGIS Repository RCA Corporation Government Systems Division Mail Stop 127-327 Moorestown, NJ 08057	1
10.	Library (Code E33-05) Naval Surface Weapons Center Dahlgren, VA 22449	1

11. Dr. M. J. Gralia — 1
Applied Physics Laboratory
John Hopkins Road
Laurel, MD 20702

12. Dana Small, Code 8242 — 1
Naval Ocean Systems Center
San Diego, CA 92152

13. Estado Maior da Armada — 1
Brazilian Naval Commission
4706 Wisconsin Ave., N.W.
Washington, DC 20016

14. Diretoria de Ensino da Marinha — 1
Brazilian Naval Commission
4706 Wisconsin Ave., N.W.
Washington, DC 20016

15. Diretoria de Armamento e Comunicações da Marinha — 1
Brazilian Naval Commission
4706 Wisconsin Ave., N.W.
Washington, DC 20016

16. Instituto de Pesquisas da Marinha — 1
Brazilian Naval Commission
4706 Wisconsin Ave., N.W.
Washington, DC 20016

17. Instituto Militar de Engenharia — 1
Praia Vermelha, Urca
Rio de Janeiro, RJ
CEP 20000 , BRAZIL

18. Instituto Tecnológico da Aeronáutica — 1
São Jose dos Campos, SP
CEP 11000 , BRAZIL

19. Pontificia Universidade Católica — 1
R. Marques de São Vicente 225, Gávea
Rio de Janeiro, RJ
CEP 20000 , BRAZIL

20. Pete Wilson — 1
INMOS CORPORATION
P.O. Box 16000
Colorado Springs, CO 80935-16000

21. David May — 1
INMOS LTD.
1000 Aztec
West Almondsbury, Bristol, BS12 4SQ, UK

22. MAJ/USAF R. A. Adams, Code 52Ad — 1
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943

23. LT. Mauricio M. Cordeiro, Br. Navy — 2
Brazilian Naval Commission (DACM)
4706 Wisconsin Ave., N.W.
Washington, DC 20016

24. LCDR. Gilberto F. Mota, Br. Navy — 1
Brazilian Naval Commission (DACM)
4706 Wisconsin Ave., N.W.
Washington, DC 20016

25. LCDR. J. Vanni Filho, Br. Navy — 1
Brazilian Naval Commission (DACM)
4706 Wisconsin Ave., N.W.
Washington, DC 20016

END

JAN.

1988

DTIC